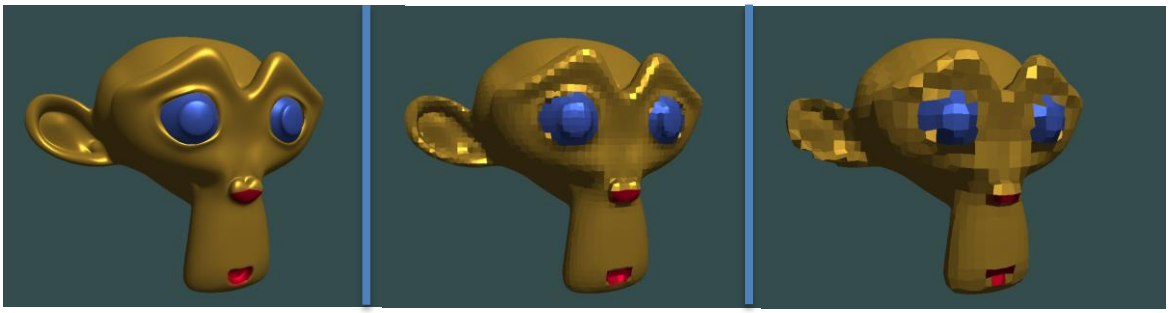
# Optimització d'Escenes per a Videojocs i Realitat Virtual

## SceneOptimizer



**David González Martos**

Director: Antonio Chica Calaf
Especialitat: Computació

Grau en Enginyeria Informàtica
Facultat d'Informàtica de Barcelona

Juny de 2018

# Contingut

# Introducció

## 1.1 Introducció

Aquest document constitueix la memòria del meu treball de fi de grau. Aquí estan explicats, en profunditat, tots els continguts que he treballat durant el desenvolupament d'aquest projecte, així com el producte final al que he arribat. La explicació és des d'un punt de vista tècnic, centrant-se en els algoritmes implementats i les eines software utilitzades.

El producte final d'aquest TFG és una aplicació executable, això vol dir que no ha sigut un projecte centrat en la recerca, sinó més aviat en el desenvolupament del codi. Tot i així, hi ha hagut un paper fonamental de recerca per poder reunir totes les tècniques utilitzades, el qual també es veurà degudament reflectit al llarg d'aquest document.

El tema del meu projecte està fortament relacionat amb els gràfics per computador, concretament amb el tractament de models 3D. Per entendre la part més tècnica de la redacció he dedicat la primera part del document a explicar els conceptes bàsics, però importants, relacionats amb aquests models.

Després ve la part on exposo amb detall els algoritmes que he investigat i implementat, comentant les seves característiques, rendiment i resultats obtinguts. Seguidament, he dedicat una secció sencera ha comentar l'aplicació final que he desenvolupat, explicant totes les funcionalitats implementades. Finalment acabo amb un anàlisi de les proves fetes i tots els resultats obtinguts com a conclusió, junt amb el material del curs de Gestió de Projectes i els annexos.

## 1.2 Objectius

Al treballar en l'àmbit dels gràfics per computador a l'actualitat, un dels aspectes més importants en el que cal centrar-se és que les aplicacions gràfiques funcionin de la manera més fluida possible. Ja sigui per videojocs, realitat virtual o augmentada, simuladors, ... És important optimitzar el seu rendiment, per millorar la experiència dels usuaris.

Hi ha molts elements que poden afectar al **rendiment**: una gran quantitat d'objectes a l'escena, càlculs molt complexos de il·luminació i ombres, físiques i col·lisions, o senzillament que un model sigui tan complex que el programa es quedi sense memòria per tractar amb tanta geometria. Una solució podria ser augmentar les prestacions hardware i adquirir una millor targeta gràfica, però aquesta alternativa no és sempre possible, a més de ser molt cara.

Indiferentment del tipus d'aplicació que s'utilitzi, una característica comuna és que totes treballen amb models, en major o menor mesura. Per aquesta raó, una bona solució general és agafar els models que seran utilitzats i passar-los per **un procés previ d'optimització per simplificar-los**. Així, al reduir la seva geometria seran més fàcils de tractar, però cal vigilar perquè evidentment també es veuran pitjor (amb menor resolució).

Aquest serà el tema amb el que em centraré principalment durant aquest projecte, la simplificació de models. El que jo proposo és **desenvolupar una aplicació gràfica** que sigui capaç de:

1. **Importar** qualsevol model o escena, indiferentment del seu format.
2. Aplicar les **simplificacions** que vulgui l'usuari, podent escollir els paràmetres i generant diferents versions optimitzades del mateix model.
3. Tornar a **exportar** el model optimitzat, amb el format desitjat.

Cal notar que ja existeixen programes de modelatge 3D que ofereixen la opció de simplificar un model. El problema que tenen és que en la majoria dels casos, com per exemple Blender, només pots simplificar els models que has creat amb el mateix programa. Què passa si has modelat una escena en un software diferent? Què passa si l'has creat en un format que ni tan sols la teva aplicació pot importar?

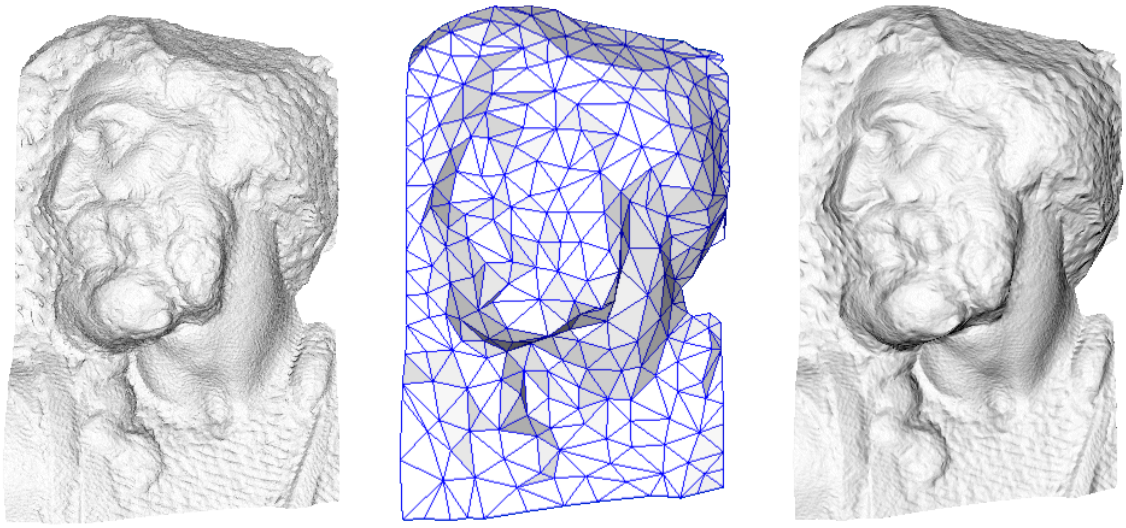
A més, en molts altres casos no permeten llibertat de poder escollir els paràmetres desitjats per l'usuari, totes les simplificacions es realitzen amb les mateixes opcions prefixades. Això és un gran problema ja que cada model és diferent, cal provar diferents tècniques i paràmetres per veure quines s'adapten millor a cada cas.

La meva aplicació intentarà unificar diferents tècniques de **reducció del detall** i solucionar tots aquests problemes, per poder simplificar pràcticament qualsevol model existent. Tot i així és important tenir en compte que el nombre de possibles inputs diferents és infinit, els models poden ser molt diferents entre sí. Que un algoritme funcioni bé per un cas, pot produir resultats dolents en altres casos. Per aquesta raó no em centraré en una única tècnica de simplificació, sinó que investigaré diferents paràmetres, millores i algoritmes. La aplicació la desenvoluparé en OpenGL 3.3, i per la interfície utilitzaré QT Designer.

A més de la reducció del detall, hi ha una altra tècnica d'optimització que estarà bé investigar: **Normal Mapping**. No la implementaré per falta de temps i perquè és força complexa, però és un tema molt interessant a tenir en compte en tot aquest procés, ja que permet afegir detall en un model sense incrementar la seva geometria (ho fa a través de la textura).

D'aquesta manera, el procés ideal per optimitzar un model seria primer reduir la seva geometria, i després afegir detall a la seva textura per fer que es vegi lo més similar possible al model original (però mantenint la poca geometria).

La següent imatge de Paolo Cignoni mostra exactament el resultat d'aquest procés:



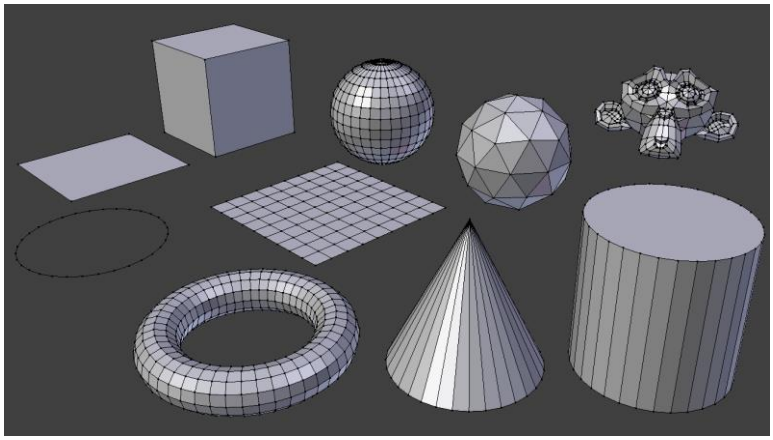
| Model original<br>4 milions de triangles | Model simplificat<br>500 triangles | Model simplificat +<br>Normal Mapping<br>500 triangles |
|---|---|---|

## Models

Abans d'entrar a simplificar models, primer necessitem saber què és exactament un model i què pot contenir. Si el lector no coneix aquest tema, aquesta introducció serà bona per repassar els conceptes bàsics sobre gràfics que es tractaran en el document. Si el lector coneix el tema també és recomanable llegir aquest apartat perquè explicaré com treballaré amb els models i com els importaré. Part d'aquesta explicació ha sigut realitzada amb l'ajuda de la web LearnOpenGL.

S'ha d'entendre com a model a un objecte individual, sigui quina sigui la seva forma. Així, una **escena** conté un o més **models**, i un model conté un o més **meshes** (malles). Un model en una escena té sempre una posició, una orientació i una escala.

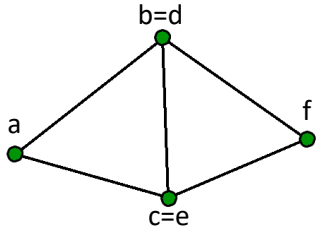
Els meshes són parts diferenciables dins d'un mateix model, que poden estar ordenades de manera jeràrquica. Per exemple, podríem tenir un model d'una figura humana amb 6 meshes: cap, cos, els dos braços i les dues cames. Un mesh està format per cares, i una cara està formada per vèrtexs.



Exemple d'una escena amb algunes figures primitives de Blender.

En aquest cas cada model conté només un mesh.

Les cares normalment tenen 3 o 4 vèrtexs. Com que el desenvolupament del meu projecte serà en OpenGL, treballaré sempre amb **triangles**. Seria lògic dir que els models sempre tindran un nombre de vèrtexs múltiple de 3, però això no és cert. Una optimització que es fa per reduir el nombre de vèrtexs repetits és utilitzar **índexs**: enters que apunten a vèrtexs. És a dir, per a aquesta figura:




Tenim 2 cares, 4 vèrtexs i 6 índexs (la primera cara està formada per els índexs a,b,c i las segona per d,e,f). Així ens estalviem 2 vèrtexs, que ocupen molt més en memòria que 2 índexs. Per tant, el número de índexs sí que serà múltiple de 3.

## 2.1 Vèrtexs

Ara que sabem que una escena està formada per models, els models per meshes, els meshes per cares, i les cares per vèrtexs, anem a veure la part més interessant on està realment continguda tota la informació del model.

Els vèrtexs són la unitat atòmica d'un model. Contenen tots els atributs necessaris, els més importants són:

**Posició.** Vector de 3 floats que representa les coordenades x,y,z del vèrtex (en el sistema de coordenades local). Cal notar que un vèrtex no té ni orientació ni escala, ja que no té sentit parlar d'aquests conceptes per un punt, però el model sí que en té.

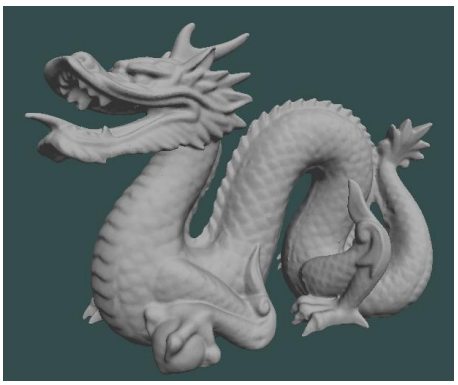
**Normal.** Vector de 3 floats que representa la direcció de la normal del vèrtex. És important per als càlculs d'il·luminació. Un model por venir representat amb les normals per cara (perpendiculars al pla de la cara) o per vèrtex (sol ser la mitjana de les normals de les cares incidents al vèrtex). En aquest projecte utilitzaré normals per vèrtex (ja que no guardo informació dels triangles).

**Tex Coords.** Són les coordenades de textura UV. Vector de 2 floats entre 0 i 1 que representa el punt de la textura on queda assignat el vèrtex. Només es fan servir si el model utilitza una textura.

**Tangents i Bitangents.** Son dos vectors de 3 floats cadascun, ortogonals entre si i amb la normal (formen $90^{o}$ entre ells). Són molt útils per al càlcul de Normal Mapping, per millorar la il·luminació.

Aquest és un bon moment per demostrar que la simplificació de models pot arribar a ser molt necessària, tenint en compte quant poden arribar a ocupar.

Suposem que tenim el següent model, que utilitza tots els atributs descrits anteriorment, carregat en un videojoc. Té 437.645 vèrtexs i 2.614.242 índexs. Si tenim en compte que tant enters com floats ocupen 4 bytes en memòria, llavors:



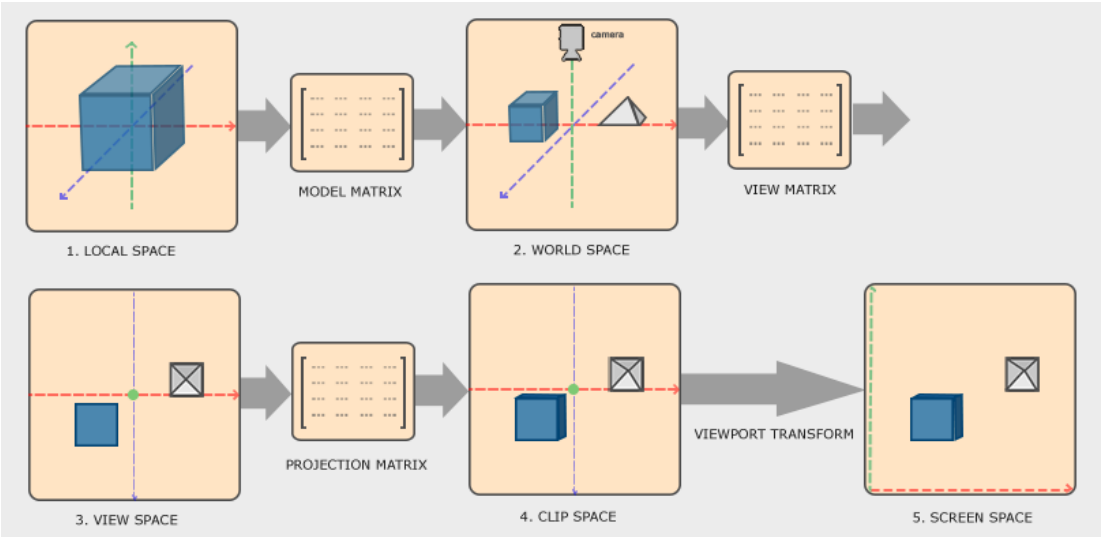
1 vèrtex ocupa 12 + 12 + 8 + 24 = 56 bytes

56 * 437.645 + 4 * 2.614.242 = 34.965.088 bytes

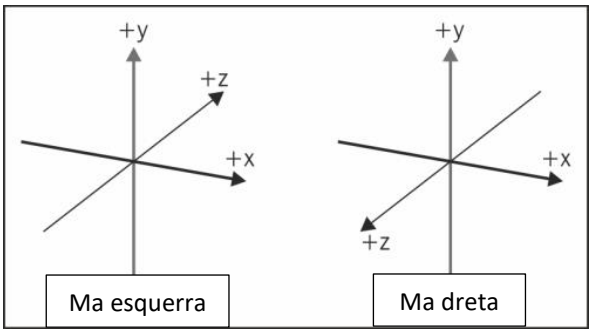
És a dir, aquest únic model ocupa casi 35 MB en memòria. És exagerat, tenint en compte que tota aquesta geometria es renderitza a cada fotograma. Imagina a més que hi ha 20 enemics invocats amb aquest model, més tots els altres elements de l'escena, amb el possible càlcul de les seves col·lisions, il·luminació, ... És completament inviable.

## 2.2 Sistema de Coordenades i Transformacions

Anem a parlar ara dels sistemes de coordenades en els que es poden representar els models. A l'anterior apartat he comentat que la posició d'un vèrtex venia en funció del sistema de coordenades local, també anomenat **Object Space** o Local Space. En aquest sistema, totes les posicions venen descrites utilitzant el model com a referencia, però a l'hora de renderitzar-les necessitem assignar-les la posició d'un píxel (**Screen Space**, és a dir, sistema de coordenades de la pantalla). El procés per transformar les coordenades de Object Space a Screen Space segueix una sèrie de passos importants, on es realitzen multiplicacions de matrius per passar els vèrtexs a sistemes intermedis fins arribar al final. Aquest procés està explicat amb detall a la web de LearnOpenGL, no cal comentar els detalls aquí.




Un altre tema a tenir en compte és que OpenGL utilitza el sistema de la ma dreta. Això vol dir que l'eix de les "x" apunta a la dreta, l'eix de les "y" cap amunt, i l'eix de les "z" és perpendicular a la pantalla, com si sortís cap a tu.




És important saber això perquè altres programes poden utilitzar el sistema de la ma esquerra, que té l'eix de les "z" invertit, i pot causar que a l'importar models estiguin completament girats.

Per finalitzar aquest apartat, comentar també que per modificar la posició, rotació i escalat d'un model s'utilitzen transformacions geomètriques. Consisteixen en multiplicar les posicions dels vèrtexs per matrius específiques. Les que jo utilitzo són les següents:

- **Translació**. Consisteix en desplaçar la posició del vèrtex en una direcció T. A la meva aplicació faig una translació inicial quan s'importa un model, per col·locar-lo centrat a l'origen de coordenades (0,0,0), ja que els càlculs numèrics funcionen millorar a prop del 0 (per temes de precisió).

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

*Translació d'un vèrtex en la direcció T*

- **Rotació**. Consisteix en rotar el vèrtex un cert angle θ en un dels tres eixos. A la meva aplicació l'usuari té l'opció per rotar el model en qualsevol direcció. Útil sobretot si el model apareix d'esquena (girat per culpa del sistema de la ma dreta).

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

*Rotació d'un vèrtex en l'eix x, amb angle* θ

- **Escalat**. Consisteix en modificar la llargada del vector posició en un factor S. A la meva aplicació faig un escalat inicial quan s'importa un model, per fer que les dimensiones s'adaptin als paràmetres de la càmera.

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

*Escalat d'un vèrtex amb factor S*

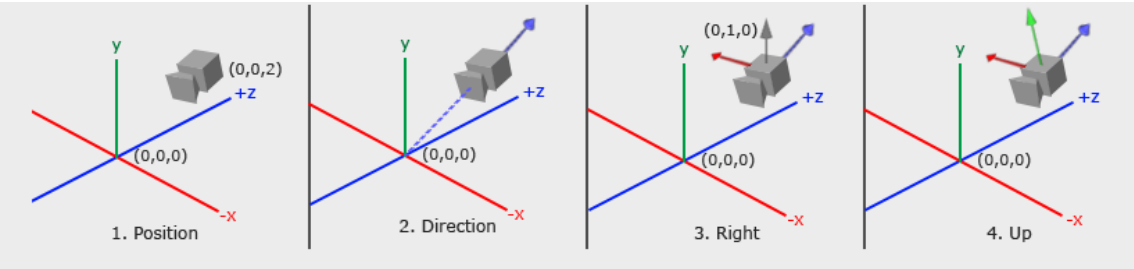
## 2.3 Altres elements de l'escena

Apart de models, una escena conté altres elements per la seva correcta renderització. Els dos més importants que utilitzo a la meva aplicació són càmera i il·luminació:

**Càmera.** Tinc un objecte que simula l'existència d'una càmera a l'escena de tipus FPS (First Person Shooter), és a dir, una càmera perspectiva en primera persona que pots moure de manera lliure per l'espai 3D. Atributs d'aquest objecte:

- **Posició**. A l'igual que un model qualsevol, té una posició representada per un vector de 3 floats, en el sistema de coordenades de World Space. Posició i orientació són modificables amb el botó dret del ratolí i les tecles W,A,S,D.

- **Orientació**. La orientació de la càmera ve definida per 3 vectors de 3 floats cadascun, ortogonals entre ells (formen 90º): Direction, Right i Up. L'usuari també podrà modificar l'orientació de manera lliure amb el ratolí.

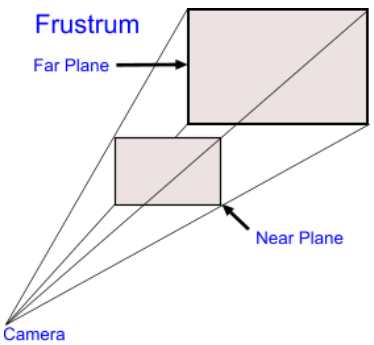



- **Velocitat**. Determina la velocitat de moviment per l'espai 3D. Modificable a través de la interfície gràfica de l'aplicació. Prement la tecla Ctrl i Shift alenteixes i acceleres respectivament la velocitat.

- **Zoom**. Determina l'angle d'obertura de la càmera (la distància a la que es veuen tots els objectes). Modificable amb la rodeta del ratolí.

- **Tamany de la pantalla**. Valor prefixat de les dimensions de la pantalla, amb una amplada de 800 píxels i una altura de 600 píxels.

- **Znear / Zfar**. Determinen les distàncies mínima i màxima, respectivament, a la que un objecte queda dins del frustum (i per tant, es renderitza).
  Valors prefixats a Znear = 0.001, Zfar = 1000



**Il·luminació.** A la meva aplicació també faig un càlcul d'il·luminació simple per poder veure els models correctament. Utilitzo una **llum direccional** amb els següents paràmetres:

- **Direcció.** Com que és una llum direccional no té sentit parlar de posició (és com si estiguéssim parlant del sol, que realment sí que té una posició però està molt allunyada) sinó de direcció. És un vector de 3 floats inicialitzat al principi de tal manera que apunta a la part frontal del model. Es pot modificar des de la interfície gràfica.

- **Ambient.** La component ambiental simula de manera simple la il·luminació global de l'escena. És una constant petita que se suma al càlcul final de la llum. Vector de 3 floats fixat a (0.2, 0.2, 0.2).

- **Diffuse.** La component difusa dona a una superfície més il·luminació quan més alineat està amb la direcció de la llum. Vector de 3 floats fixat a (0.5, 0.5, 0.5).

- **Specular.** La component especular dona la propietat de reflexió, tenint en compte la direcció en la que està mirant la càmera. Vector de 3 floats fixat a (1, 1, 1).

- **Shininess.** La component de shininess indica la magnitud de la taca especular i la seva dispersió en una superfície. Float inicialitzat a 32.

En OpenGL, els càlculs de la il·luminació es fan en els shaders. Els shaders són petits programes de la GPU que s'executen durant el pipeline gràfic. Utilitzo els dos shaders bàsics: vèrtex shader i fragment shader. Els atributs anteriors es passen als shaders com a uniforms, i en el fragment shader faig els càlculs necessaris per generar il·luminació de **Blinn – Phong** (és un algoritme que genera bons resultats sobretot amb les reflexions especulars).

*Model sense il·luminació*

*Model amb il·luminació Blinn-Phong*

## 2.4 Importar / Exportar. ASSIMP

Per acabar aquesta secció explicaré com tractaré amb els models de manera externa. He explicat que quan tenim un model carregat conté meshes, vèrtexs, atributs, etc. Que estan carregats en memòria, però com es guarda tota aquesta informació de manera externa i com s'importa al nostre programa?

Els models es guarden en fitxers de definició de geometria, guardant els valors de tots els atributs com una seqüència de dades. El format més utilitzat és Wavefront Object (.obj), però hi ha altres com Collada (.dae) o l'utilitzat per els models de Stanford (.ply).

A més, aquests fitxers poden venir acompanyats d'altres fitxer que guarden els materials del model o altra informació complementaria, com els fitxers MTL (.mtl) que acompanyen sempre als .obj.

Per tractar amb aquest tema de la manera més ràpida possible, i tenint en compte que cada format és diferent i no tinc temps d'implementar un importador específic per cada possible format, utilitzaré una llibreria externa anomenada **ASSIMP** (Open Asset Import Library). Aquesta llibreria té implementades les funcions per importar i exportar models tractant amb un ampli ventall de formats. De fet, l'importador accepta fins a 39 formats diferents. Així puc automatitzar aquest procés i centrar-me més en els algorismes de simplificació. Llibreria compilada amb CMake i MinGW.

Quan Assimp importa un model i llegeix tot el seu contingut, guarda la informació en una estructura interna anomenada **aiScene**. Aquesta és un arbre accessible a través dels seus nodes (via punters). D'aquí es poden extreure tots els elements que necessitis per guardar-los en les teves pròpies estructures de dades.

Per exportar funciona igual. Un cop has modificat els atributs com vols en el teu programa, crees una nova aiScene i li col·loques tota la informació. Llavors crides a la funció per exportar aquesta aiScene, també amb el format que vulguis. ASSIMP genera automàticament tots els arxius necessaris.

Per acabar, quan importes un model amb ASSIMP pots passar-li un conjunt de **flags** per aplicar algunes opcions de post-processat. En el meu programa utilitzo les següents:

- **aiProcess_Triangulate**. Fa que totes les cares siguin triangles. És molt útil, ja que OpenGL només treballa amb triangles, i molts cops hi ha models que venen amb cares de mes de 3 vèrtexs.
- **aiProcess_FlipUVs**. Inverteix les coordenades de textura, necessari perquè OpenGL necessita llegir-les així.
- **aiProcess_GenSmoothNormals**. Si el model no conté normals, genera automàticament normals suaus per cada vèrtex.

## Simplificació de models

Ara ja sabem què és un model, quins atributs conté, el tenim importat i carregat en les nostres estructures de dades. Ha arribat el moment de processar-lo.

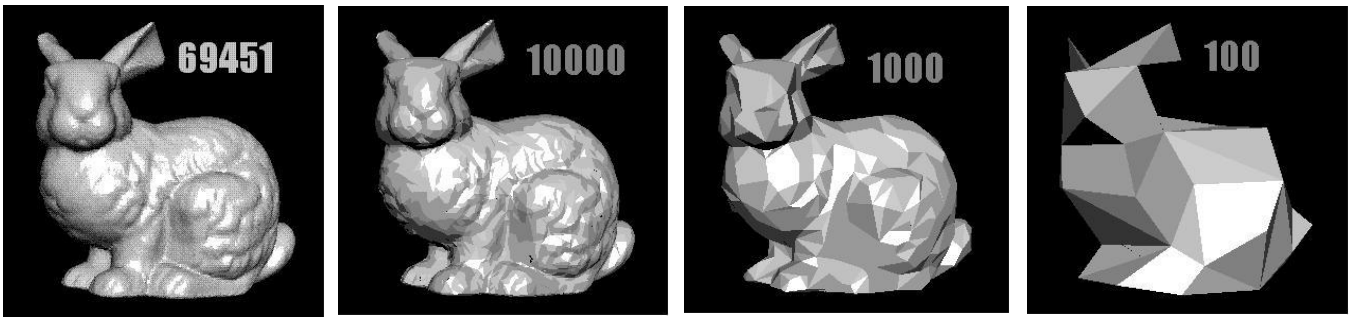
El gran objectiu d'aquest projecte és la simplificació de models, és a dir, la reducció de la geometria. L'única manera d'aconseguir això és reduir el nombre de vèrtexs que conté el model (i que indirectament també redueix el nombre de índexs i cares que utilitza).

Evidentment, s'ha d'intentar fer amb el menor error possible, mantenint al màxim la forma original del model i la seva estructura. Tot i així, és inevitable que els detalls més petits es perdin en el procés. Ara bé, hi ha dos possibles escenaris en els quals vulguem simplificar un model:

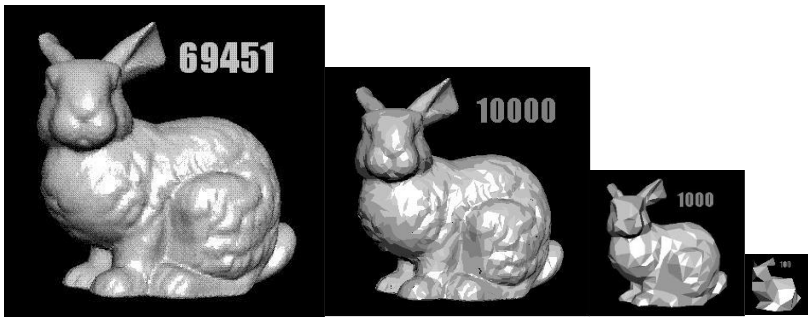
1. A partir del model original volem obtenir un únic model representatiu amb molta menys geometria, que substituirà en tot moment a l'original.
2. A partir del model original volem obtenir una sèrie de models, cadascun més simplificat que l'anterior, anomenats **LODs** (Levels Of Detail, o Nivells de Detall). Aquests LODs es canviaran segons convingui.

Com ja es pot deduir, el primer cas és més simple ja que substituïm completament el model original. El problema ve quan t'apropes molt a aquest model simplificat, ja que notaràs clarament la pèrdua de detall.

Amb la segona opció, el propòsit és anar canviant el LOD que renderitzes segons la distancia de la càmera. Si estàs molt a prop, utilitzes el model original (que seria el LOD 0). A mesura que t'allunyes, vas canviant al LOD 1, LOD 2, ... i cada cop vas perdent detall, però no es notarà ja que cada cop estarà més lluny. Aquest mètode és una mica més complex, però renderitzar el LOD en funció de la distància definitivament donarà millors resultats. El principal problema és intentar que el canvi no es noti: per exemple, una opció és interpolar els dos LODs consecutius perquè el canvi no sigui tan abrupte. Tot això ho comprovarem amb més detall a la secció de proves i resultats.




*LOD 0* *LOD 1* *LOD 2* *LOD 3*



*Si es renderitza el LOD en funció de la distància no es nota tant la pèrdua del detall*

Pel que fa als algoritmes, hi ha dos grans tècniques de simplificació en les que em centraré: **Vertex Clustering** i **Edge Collapse**.

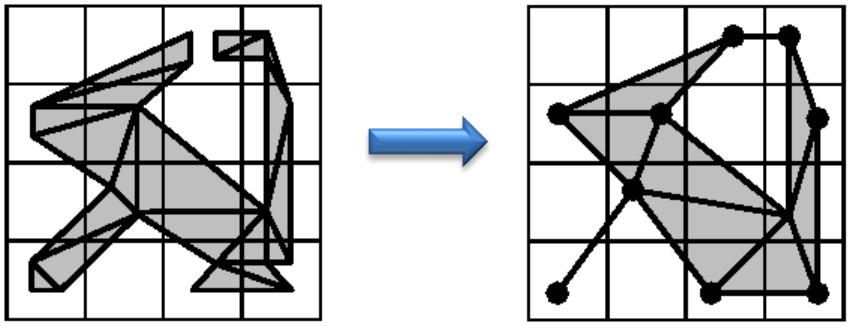
Les dues tenen el mateix objectiu, reduir els vèrtexs del model, però cadascuna parteix d'una idea diferent a l'hora de tractar amb la distància entre vèrtexs i les seves unions. Serà molt interessant comparar els punts forts i febles entre els dos, com de robustos són (funcionen per qualsevol model, o necessiten que els models tinguin alguna característica?), i la seva eficiència tant en temps com en espai.

De totes maneres, que un algorisme sigui lent a l'executar-se no és un factor crític. Normalment aquest procés de simplificació és un pas previ, de preprocessat. Un cop tens el model simplificat, el pots utilitzar sempre que vulguis sense necessitat de tornar a fer els càlculs: si cal esperar un temps al principi, però només un cop, no és un problema.

## 3.1 Vertex Clustering

Comencem per l'algoritme més important, que he implementat sencerament i li he dedicat més temps.

La idea de Vertex Clustering és ajuntar aquells vèrtexs que estan lo suficientment a prop entre ells, i així formar clústers (grups de vèrtexs). Per aconseguir això, es genera una graella, i tots els vèrtexs que cauen dins una mateixa cel·la s'ajunten en un únic vèrtex representatiu. És simple però efectiu.



Els principals avantatges d'aquesta tècnica són el seu reduït cost computacional i l'alta taxa de reducció de dades. A més, com podrem comprovar, és molt robust podent simplificar pràcticament qualsevol model. Per aquesta raó ha sigut el primer algoritme que he implementat.

Com a contrapunt, és difícil controlar el nombre de triangles que tindrà el model simplificat i té molta dificultat a l'hora de preservar els detalls, a més de ser insensible a la connectivitat entre vèrtexs (ignora completament les arestes).

També li donarem molta importància als atributs dels vèrtexs, ja que al ajuntar un grup d'aquests s'han de generar nous atributs pel vèrtex representatiu, i han de mantenir les possibles discontinuïtats que puguin haver.

L'algoritme es pot dividir en tres grans passos:

1. Creació de la graella virtual i assignació de cada vèrtex original a una cel·la.
2. Per cada cel·la ocupada, creació del nou vèrtex representatiu.
3. Eliminació dels triangles degenerats.

Anem a analitzar-los amb detall.

### 3.1.1 Graella virtual

Quan parlem de generar una graella virtual, s'ha de vigilar amb l'espai que pot arribar a ocupar. Evidentment, aquesta graella tindrà una **resolució** (serà el nombre de cel·les en les tres direccions) que podrà ser modificada des de l'aplicació gràfica.

Sigui "g" la resolució de la graella. Si creem una matriu cúbica de mida g, llavors aquesta ocuparà un espai de $\theta(g^3)$. Per resolucions petites (que impliquen simplificacions grans) pot ser una mida controlable, però a la que pugem una mica es torna desmesurat. Per g = 256, que sol ser una resolució habitual, acabaríem amb una matriu de casi 17 milions de posicions, una mida totalment inviable sobretot tenint en compte que la gran majoria de cel·les estaran buides.

És cert que l'accés és molt ràpid, però cal buscar una alternativa a les matrius convencionals per solucionar els problemes d'espai. Algunes opcions serien utilitzar maps o hashmaps, que generen posicions noves en funció de l'espai que necessiten. En el cas dels hashmaps caldria generar una bona key per assignar les posicions, evitant en la mesura de lo possible les col·lisions (cel·les que acabessin amb la mateixa key).

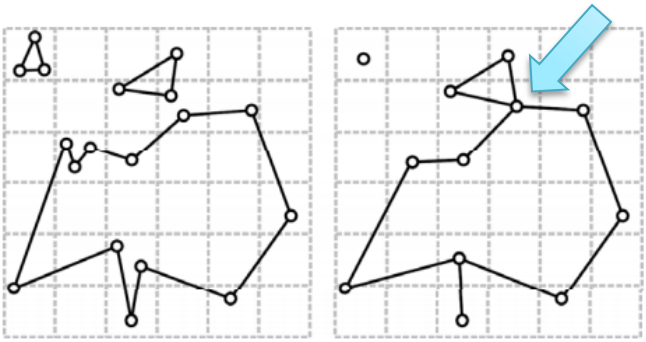
Per simplificar les coses, he decidit implementar la graella virtual amb un **map**. La key és única per cada cel·la, determinada per la posició del vèrtex i la resolució de la graella (explicat a continuació). Així, la mida que ocuparà aquesta graella serà directament proporcional al nombre de vèrtexs output que tindrà el model simplificat, i no hi haurà espais buits. No perdem eficiència temporal, ja que l'accés a un map segueix sent constant.

Per calcular la key determino a quina cel·la acaba un vèrtex donat, a partir de la seva posició (que està escala a [0,1]) i la resolució de la graella:

```
int cellX, cellY, cellZ;
string key;
cellX = int(vertex.x * gridResolution);
cellY = int(vertex.y * gridResolution);
cellZ = int(vertex.z * gridResolution);
key = string(cellX) + "0" + string(cellY) + "0" + string(cellZ);
grid[key].push_back(vèrtex.id);
```

Per la key concateno les tres coordenades, amb un 0 entremig. És molt important posar un element separador entre cada coordenada per evitar que dos vèrtexs molt allunyats acabin en la mateixa cel·la (per exemple, el vèrtex (5,3,41) acabaria en la mateixa cel·la que el vèrtex (53,4,1) amb la key "5341").

En aquest procés es pot veure clarament la limitació comentada sobre la connectivitat. Hi haurà casos en els que vèrtexs que no comparteixen cap connexió acabin ajuntats en la mateixa cel·la, creant una connexió abans inexistent.



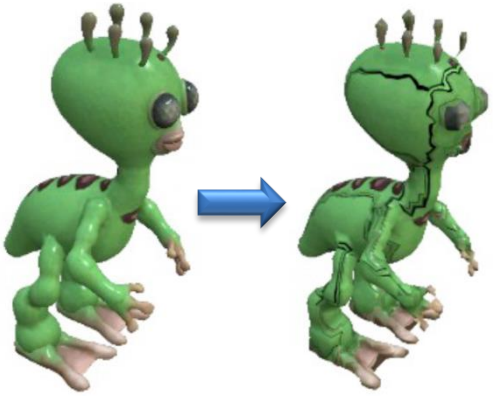
Un cop arribats a aquest punt, ja tenim una graella virtual construïda on cada posició representa una cel·la (ocupant el menor espai possible), i cada cel·la conté una llista amb tots els vèrtexs del model original que han caigut aquí.

### 3.1.2 Nous vèrtexs

Ara toca crear un vèrtex representatiu per cada cel·la. Els atributs d'aquest nou vèrtex es calcularan a partir dels atributs de tots els vèrtexs que han caigut en la cel·la, però hi ha diferents maneres de fer això:

1. **Escollir un vèrtex** i utilitzar els seus atributs. El problema és, com escollim el millor vèrtex? Més endavant veurem una mètrica per calcular els errors de cada vèrtex, anomenada **Quadric Errors**, que ens ajudarà en aquesta selecció.
2. **Fer la mitjana** de tots els atributs. Aquest ha sigut la primera aproximació que he implementat en la aplicació gràfica.

Per les posicions i les normals, calcular la mitjana funciona força bé. El gran problema apareix amb les coordenades de textura:



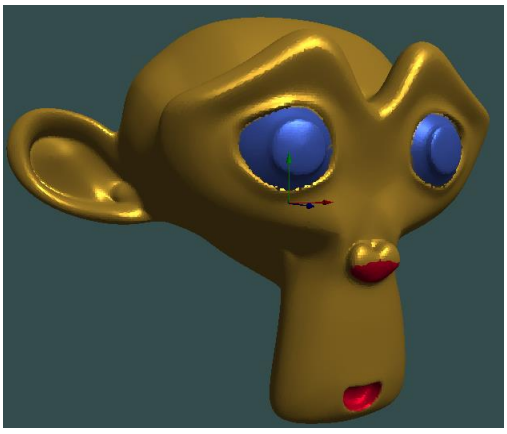
*Textura que presenta discontinuïtats*

Es pot apreciar com apareixen unes línies estranyes en el model simplificat. Aquests artifacts es deuen a que les coordenades de textura tenen discontinuïtats, causades al fer el unwrap del mesh. Si s'intenta fer la mitjana entre dues coordenades de textura, et surt una coordenada errònia que pot acabar on no hi ha res o en mig d'un altre tros.

El que faig per la meva aplicació és una combinació:

- Per les posicions calculo la mitjana. El resultat és força consistent i correcte. Més endavant ho milloraré amb Quadric Errors.
- Per les coordenades de textura creo tants vèrtexs duplicats com discontinuïtats hi hagi. Això causa que una cel·la pugui acabar amb més d'un vèrtex, però l'increment és molt petit i la millora molt notòria. Aprofito els duplicats per agafar també les normals.
- Per les tangents i bitangents no cal fer res, ja que les calcularé posteriorment quan facin falta (per Normal Mapping).

*Simplificació amb Tex Coords mitjanes*



*Simplificació amb Tex Coords duplicats*

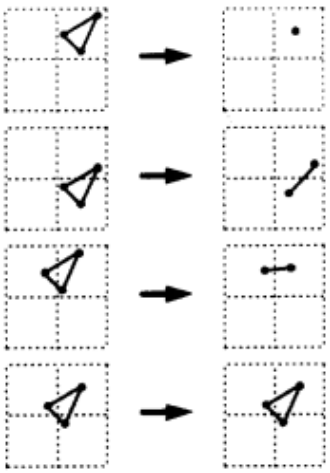
Apart de la preservació de discontinuïtats en atributs, hi ha altres optimitzacions que es poden aplicar però que no implementaré, ja que només aporten petites millores en casos molt concrets i són molt complexes (em portaria molt de temps, i prefereixo centrar-me en tècniques més generals). Algunes d'aquestes millores són la preservació de les formes més petites que la mida d'una cel·la, com podria ser el bigoti d'un gat o les fulles dels arbres, i la preservació dels ossos en les animacions, per evitar que una simplificació es vegi diferent depenent de cada frame de la animació. Aquestes tècniques estan ben explicades en el paper de Rapid Simplification of Multi-Attribute Meshes [3].

### 3.1.3 Triangles degenerats

Un cop tenim tots els vèrtexs nous que hi haurà en el model simplificat, cal crear els triangles i veure si alguna cara a desaparegut.

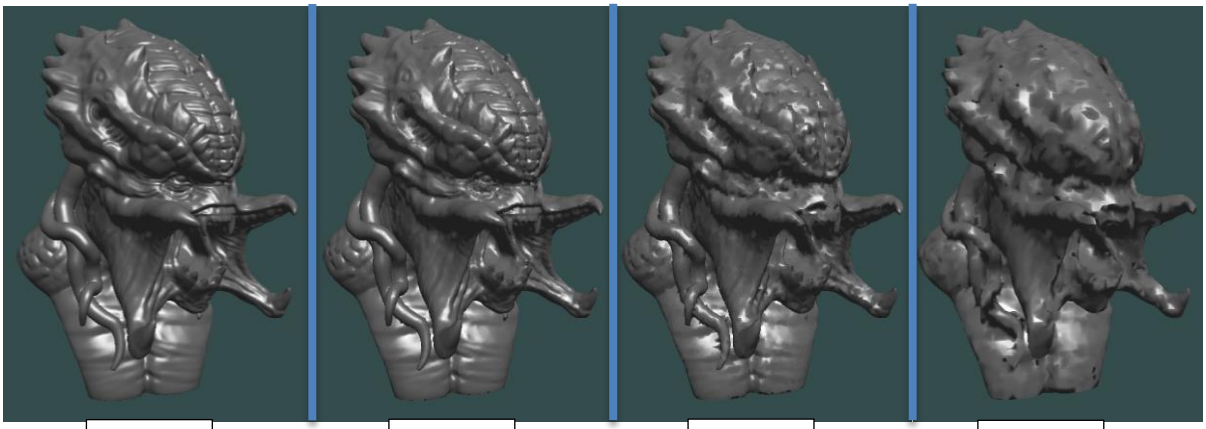
Depenent de les cel·les on hagin caigut els vèrtexs inicials, poden haver tres escenaris:

- Els 3 vèrtexs d'un triangle han caigut en la mateixa cel·la. Com que els 3 han sigut substituïts per un únic vèrtex, el triangle a desaparegut completament.
- Han caigut 2 vèrtex en una cel·la i un vèrtex en una altra. Com que hi ha dues cel·les en ús, el resultat és una línia que connecta dos vèrtexs, però el triangle ha desaparegut.
- Cada vèrtex ha caigut en una cel·la diferent. En aquest cas hi ha 3 línies, que connecten les 3 cel·les mantenint el triangle.



Com es pot veure, només s'han de preservar aquells triangles tals que els seus vèrtexs han acabat en 3 cel·les diferents. La resta són triangles degenerats convertits en punts o línies, que s'han d'eliminar (senzillament ignorem les tripletes de índexs).

Amb la nova llista de vèrtexs i índexs ja tenim tot el material necessari per crear el nou model simplificat. Els resultats visuals els veurem en detall a la secció de proves i resultats per comparar-los amb els altres algoritmes, amb ajuda de tots els models de test que he obtingut.



*100%* *45%* *16%* *5%*

### 3.1.4 Rendiment

Per acabar analitzaré el cost d'aquest algoritme. Un aspecte interesant és la relació entre el nivell de simplificació, la resolució de la graella i el temps:

Quan **més gran** és el nivell de simplificació (menys vèrtexs queden en el model final), la resolució de la graella és **més petita** i el temps **menor**. Sembla contra intuïtiu el fet que simplificar més costi menys, però té sentit perquè el cost està directament relacionat amb el nombre de vèrtexs output. En aquest sentit Vertex Clustering és l'oposat a Edge Collapse, que triga més quan més simplifica.

Sigui n el nombre de vèrtexs del model original, m el nombre de vèrtexs del model simplificat, k el nombre de índexs originals. Si tenim que k >= n >= m, el cost de cada pas de l'algoritme és:

| | **Cost en espai** | **Cost en temps** |
|---|---|---|
| **Graella virtual** | θ(m) | θ(n) |
| **Nous Vèrtexs** | θ(m) | θ(m) |
| **Triangles degenerats** | θ(k) | θ(k) |
| **Global** | **θ(2m + k)** | **θ(n + m + k)** |

El cost computacional és molt baix: és **lineal** en relació a la quantitat de geometria del model original i simplificat, tant en temps com en espai.

Aquest algoritme l'he implementat completament en la meva aplicació gràfica (excepte les millores especials del paper de Rapid Simplification). Al executar-lo amb tots els models de prova que tinc (de mides molt variades, fins a 4 o 5 milions de vèrtexs), els resultats són altament satisfactoris. Per la gran majoria dels casos triga menys d'un segon en executar-se completament.

I com he comentat al principi és molt robust. Passis el model que li passis el simplificarà, dona igual com estigui modelat. A més de tenir una alta taxa de reducció de dades, podent ajustar el paràmetre de la resolució de la graella des de l'aplicació gràfica per crear de manera personalitzada diferents LODs del mateix model, cada un més simplificat. No té límit, però evidentment si vas simplificant sense parar arriba un punt que el model final és un cub sense forma. És treball de l'usuari decidir fins a on vol arribar.

## 3.2 Edge Collapse

Amb Vèrtex Clustering hem vist un algoritme ràpid i robust, però que no treballa bé amb detalls petits. Edge Collapse és justament el contrari: és una tècnica amb alt cost computacional, poc robust, però que preserva millor la forma i els detalls.

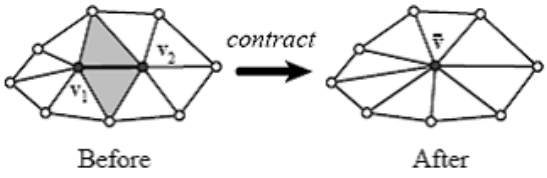
He decidit no implementar Edge Collapse en la meva aplicació, degut a que és molt més lent i no funciona per tots els models (i per tant la majoria dels casos s'utilitzaria Vertex Clustering). Si aquest projecte no durés 4 mesos i tingués el temps limitat, sens dubte seria molta interessant implementar-lo. De totes maneres l'analitzaré igual, per veure la simplificació des d'un altre punt de vista.

Edge Collapse, també anomenat **Progressive Meshes**, va ser introduït per Hugues Hoppe al 1996 i existeix un paper que explica amb detall tot aquest procés [2].

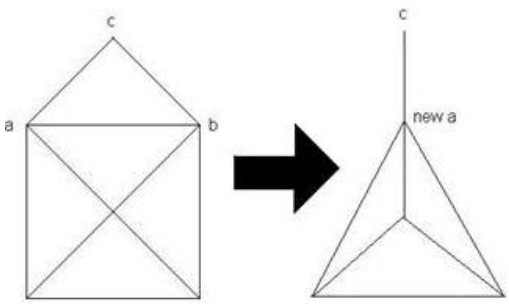
Ara, en comptes de centrar-se en els vèrtexs i ajuntar-los, Edge Collapse treballa amb les arestes per col·lapsar-les. Les arestes són les connexions entre vèrtexs, pet tant ara es respectarà molt més l'estructura del model. Ja no passarà que dos vèrtexs separats, que no comparteixen cap aresta, acabin ajuntats.

### 3.2.1 Tècnica

La idea és iterar sobre el model original i en cada pas eliminar una aresta, col·lapsant els dos vèrtexs incidents a la aresta i eliminant 2 triangles de mitja en cada volta.




Aquest pas s'anomena **contracció**. En general, una contracció només es fa si és valida. Per exemple, en la següent figura no seria desitjable fer la contracció entre a i b, ja que el triangle degeneraria en una línia que no té gaire sentit, i perdria la forma original.



Un altre detall important és que quan es realitza una contracció, tot l'entorn proper a aquesta aresta es veu afectat i canvia, per tant s'han d'actualitzar els atributs dels vèrtexs afectats i canviar els índexs d'alguns triangles. Aquí ja podem començar a veure la complexitat computacional que presenta.

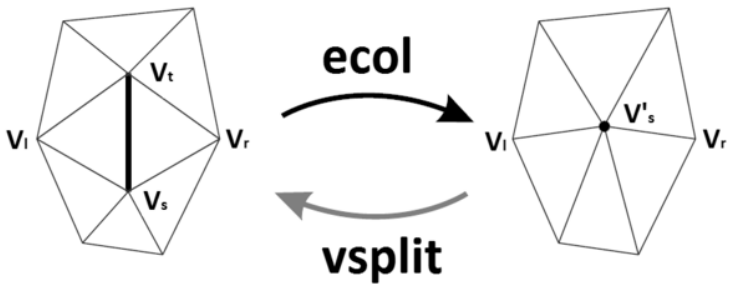
Ara la pregunta és, com seleccionem la aresta que volem col·lapsar a cada iteració, de les milions que té el model? Per suposat hi ha diferents maneres de fer-ho. La més simple és seleccionar sempre la aresta més curta, però això implica haver de guardar les distàncies de totes les arestes i anar actualitzant-les en cada volta, un cost addicional afegit. Si volem anar més llunys també podríem assignar pesos.

En el següent apartat veurem que la millora de **Quadric Errors** és una de les millors opcions per seleccionar la aresta que té el menor error, es a dir, que preserva millor la figura del model.

Un cop tenim la aresta seleccionada que volem col·lapsar, necessitem determinar el nou vèrtex que substituirà els dos de l'aresta. Semblant a Vertex Clustering, podem fer mitjanes, agafar un dels dos com el representatiu, etc. Un cop substituït i actualitzada la informació dels vèrtexs / índexs del voltants, ja es pot passar a la següent iteració.

Una de les grans qualitats d'aquest mètode és poder controlar amb exactitud la magnitud de la simplificació. Pots especificar el nombre exacte de polígons que vols que tingui el model final, i com que a cada iteració elimina una aresta i dos triangles l'algoritme s'aturarà quan arribi al nombre fixat. Això permet una personalització més elevada que Vertex Clustering.

Addicionalment, la idea de les contraccions també es pot utilitzar per fer el procés invers de la simplificació d'un model: augmentar i suavitzar la seva geometria. Aquesta tècnica s'anomena Vertex Split i l'utilitzen molts programes de modelatge 3D:



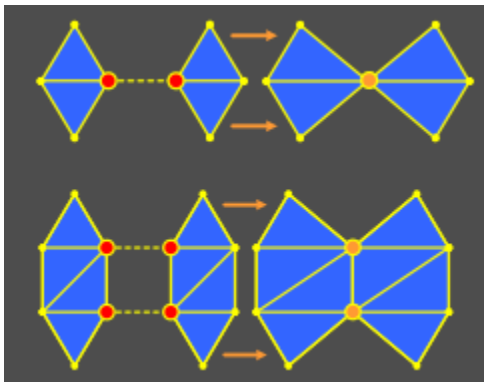
### 3.2.2 Rendiment

Per acabar amb Edge Collapse analitzarem el seu rendiment. Aquesta vegada tenim que quan **més gran** és el nivell de simplificació (menys vèrtexs queden en el model final), el temps **augmenta**, al revés que amb Vèrtex Clustering. Té sentit perquè quan més vols simplificar més iteracions cal fer, tot i que les últimes iteracions acostumen a ser més ràpides degut a que el model té menys geometria i no cal buscar entre tantes arestes.

Sigui n el nombre d'arestes del model original. El bucle extern d'iteracions tindrà un cost mig de θ(n), i dins d'una iteració cal buscar la aresta a col·lapsar per tant cal fer una altra cerca de cost θ(n). Si a més afegim un cost addicional θ(m) dins de cada iteració per actualitzar la informació dels vèrtexs / índexs propers i distàncies, tenim que el cost total és $\theta(n) * \theta(n + m) = \theta(n * (n + m)) = \theta(n^2 + nm)$.

El cost temporal de Edge Collapse és **quadràtic** en relació al nombre d'arestes del model, molt més complex que el cost lineal de Vertex Clustering. Pel que fa al cost espacial segueix sent **lineal**, però una mica més elevat que Vertex Clustering si es guarda la informació extra de les distàncies d'arestes.

Parlem ara de la robustesa. Sabem que Vertex Clustering accepta pràcticament qualsevol model sense problema, ja que no li importa la forma ni les connexions, però aquí és al contrari. Per reduir els vèrtexs necessiten estar connectats, i si no es trien bé les arestes poden aparèixer molts **artifacts** (forats en el model o formes estranyes no desitjades).



Si hi ha vèrtexs desconnectats que vols tenir en compte per les contraccions, hi ha operacions addicionals com **pair contraction** i **clúster contraction** per ajudar a ajuntar-los. Evidentment suposen un altre cost addicional a tot el procés, i si no es fa correctament pot generar encara més artifacts, o inclús generar arestes addicionals entre vèrtexs que ja estaven connectats.

Ara bé, per a aquells models ben construïts i sense elements estranys, Edge Collapse funciona excepcionalment bé per preservar els petits detalls i les formes.

## 3.3 Millora Quadric Errors

Ja hem vist les dues grans tècniques de simplificació de models: Vertex Clustering i Edge Collapse. De les dues, he decidit implementar únicament per la meva aplicació Vertex Clustering degut als avantatges que ofereix respecte Edge Collapse: és més ràpid i robust, i hem permet utilitzar-lo amb tots els models que tinc de prova.

Això si, durant les dues explicacions he mencionat la millora de **Quadric Errors**. Aquesta és una tècnica molt interessant que ajuda a millorar les posicions dels nous vèrtexs, triant sempre la posició òptima tal que preserva la forma del model original, en la mesura de lo possible. Introduïda per Garland i Heckbert, en el seu paper s'explica amb detall [5].

Lo bo que té Quadric Errors es que serveix tant per Vertex Clustering com per Edge Collapse, i per qualsevol algoritme de simplificació en general. Com que és tant útil, i no és massa complexa, he decidit implementar aquesta millora en la meva aplicació.

Evidentment, en l'aplicació l'usuari podrà decidir si vol utilitzar aquesta millora o no. De fet no sempre funciona bé, veurem en la secció de proves i resultats que per alguns models té comportaments estranys.

### 3.3.1 Tècnica

L'objectiu de Quadric Errors és triar les posicions òptimes dels nous vèrtexs per mantenir la forma original del model. Per aconseguir això, es realitzen una sèrie de càlculs previs amb matrius per guardar els errors de cada vèrtex, tenint en compte les cares a les que són incidents i aprofitant la descripció matemàtica de que un vèrtex ve definit per la intersecció d'un conjunt de plans. Per aquesta explicació utilitzaré conceptes d'àlgebra lineal.

**Passos de l'algoritme:**

1. Primer es fa un recorregut per tots els triangles del model original.
   Per cada triangle es troba la seva equació del pla, de la forma:
   $\boldsymbol{ax + by + cz + d = 0}$ on $\boldsymbol{a, b, c, d}$ són constants i $\boldsymbol{a^2 + b^2 + c^2 = 1}$
   Per trobar les constants m'aprofito que sabem els vèrtexs del triangle.
   Si el triangle ve definit per els vèrtexs A, B, C, llavors la seva normal normalitzada es troba fent el següent producte vectorial:
   $\boldsymbol{n = \frac{AB \times AC}{|AB \times AC|}}$ i per definició tenim $\boldsymbol{n = \begin{pmatrix} a \\ b \\ c \end{pmatrix}}$ i $\boldsymbol{d = -A \cdot n}$

Amb aquest informació es crea la matriu simètrica $\mathbf{K_p}$ de 4x4 per cada triangle:

$$\boldsymbol{p} = [\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}]^{\boldsymbol{T}}, \qquad \boldsymbol{K_p} = \boldsymbol{p}\boldsymbol{p}^{\boldsymbol{T}} = \begin{bmatrix} \boldsymbol{a^2} & \boldsymbol{ab} & \boldsymbol{ac} & \boldsymbol{ad} \\ \boldsymbol{ab} & \boldsymbol{b^2} & \boldsymbol{bc} & \boldsymbol{bd} \\ \boldsymbol{ac} & \boldsymbol{bc} & \boldsymbol{c^2} & \boldsymbol{cd} \\ \boldsymbol{ad} & \boldsymbol{bd} & \boldsymbol{cd} & \boldsymbol{d^2} \end{bmatrix}$$

La matriu $K_p$ serveix per trobar el quadrat de la distancia entre qualsevol punt de l'espai i el pla p.

2. Sigui **P(v)** el conjunt de tots els triangles incidents al vèrtex v.
   Ara toca recórrer tots els vèrtexs. Per cada vèrtex construirem la matriu 4x4 $\mathbf{Q_v}$:

$$\boldsymbol{Q_v} = \sum_{\boldsymbol{p}}^{\boldsymbol{P(v)}} \boldsymbol{K_p}$$

   És a dir, $Q_v$ és la suma de tots els $K_p$ tals que p∈P(v). Aquesta matriu representa la posició òptima del vèrtex v, que en aquest cas és la posició actual que té v en el model original.
3. Ara que tenim les posicions òptimes per tots els vèrtexs del model original, per obtenir la posició òptima $Q_w$ d'un nou vèrtex w obtingut a partir dels altres cal sumar les seves $Q_v$.
   En el cas de Vertex Clustering tenim $\mathbf{(v_1,v_2,...,v_i) \rightarrow w, \quad Q_w = Q_1 + Q_2 + ... + Q_i}$.
   En el cas de Edge Collapse només ajuntem parelles de vèrtex per tant és encara més fàcil: $\mathbf{(v_1,v_2) \rightarrow w, \quad Q_w = Q_1 + Q_2}$.
   Un cop tenim calculat $Q_w$ necessitem treure la nova posició d'aquesta matriu. Com que volem que l'error d'aquesta nova posició sigui mínima, obtenim la següent igualtat:

$$\begin{bmatrix} \boldsymbol{q_{11}} & \boldsymbol{q_{12}} & \boldsymbol{q_{13}} & \boldsymbol{q_{14}} \\ \boldsymbol{q_{12}} & \boldsymbol{q_{22}} & \boldsymbol{q_{23}} & \boldsymbol{q_{24}} \\ \boldsymbol{q_{13}} & \boldsymbol{q_{23}} & \boldsymbol{q_{33}} & \boldsymbol{q_{34}} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \cdot \boldsymbol{w} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix} \qquad \text{On} \quad \mathbf{q_{ij}} \in \mathbf{Q_w} \quad \text{i} \quad \boldsymbol{w} = \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ \boldsymbol{z} \\ \mathbf{1} \end{bmatrix}$$

   D'aquesta manera només cal resoldre el següent sistema:

$$\boldsymbol{w} = \begin{bmatrix} \boldsymbol{q_{11}} & \boldsymbol{q_{12}} & \boldsymbol{q_{13}} & \boldsymbol{q_{14}} \\ \boldsymbol{q_{12}} & \boldsymbol{q_{22}} & \boldsymbol{q_{23}} & \boldsymbol{q_{24}} \\ \boldsymbol{q_{13}} & \boldsymbol{q_{23}} & \boldsymbol{q_{33}} & \boldsymbol{q_{34}} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{1} \end{bmatrix}$$

   I ja tindrem la posició òptima pel nou vèrtex w.

En el cas de Vèrtex Clustering, els passos 1 i 2 es fan un cop al principi i el pas 3 es fa per cada vèrtex output. Però per Edge Collapse es pot estendre encara més.
Si els tres passos els fem en cada iteració i per cada possible contracció, al final d'una iteració tenim calculades totes les possibles noves posicions w. A partir d'aquí podem veure quin és l'error de totes aquestes noves posicions, i fer la contracció que presenti l'error més baix. Aquest procés afegeix un cost computacional addicional a tot lo que ja teníem per Edge Collapse, però es converteix en la manera més precisa possible d'obtenir les millors posicions preservant la forma i els detalls del model original.

Els passos addicionals per aquest procés serien:

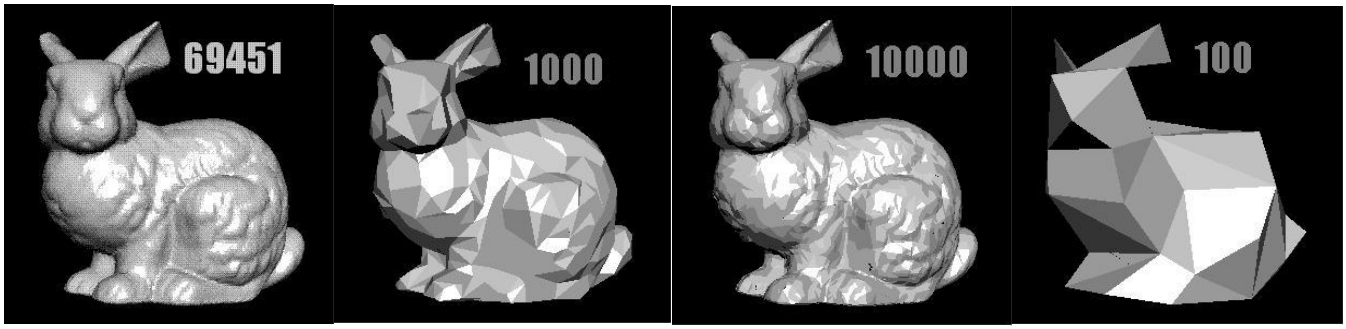
4. Per cada contracció vàlida, calculem el seu error, que equival a calcular l'error d'ajuntar parelles de vèrtexs:

   $\boldsymbol{Cost(v_1, v_2) = w^{-1} \cdot (Q_1 + Q_2) \cdot w}$ on w és la hipotètica nova posició

5. Finalment seleccionem el parell que presenta el menor cost en aquesta iteració, agafem la w calculada al pas 3 i actualitzem els costos de tots els parells afectats per el nou vèrtex. Si ho tenim tot ben actualitzat, només cal repetir el pas 5 a cada iteració fins que s'arribi al resultat desitjat.

Veurem els resultats de la combinació Vertex Clustering + Quadric Errors en la secció de proves i resultats, i com alguns models tenen problemes.

Com que Edge Collapse no l'he implementat, aquí deixo unes imatges amb el resultat de simplificar un model amb Edge Collapse + Quadric Errors, agafades d'una de les referències que utilitzo:



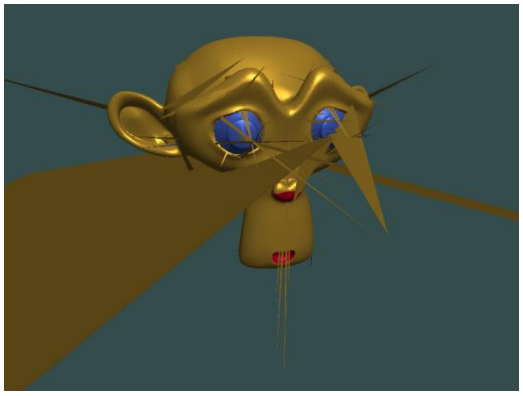

*Simplificació amb Edge Collapse + Quadric Errors. Triga 2 minuts per simplificar de 69451 a 100 cares, comparat amb els 30 ms que triga el mateix procés però amb Vertex Clustering + Quadric Errors.*

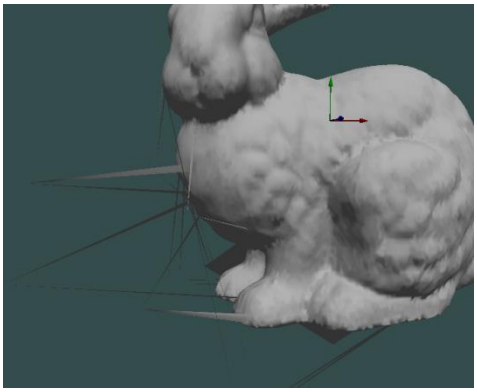
### 3.3.2 Problemes de Precisió. Eigen

Abans de passar a analitzar el rendiment de Quadric Errors m'agradaria fer un petit incís per comentar les llibreries que utilitzo per tractar dades / operacions numèriques.

A la meva aplicació, per importar i tractar amb els models utilitzo la llibreria GLM, que permet manipular vectors i matrius de manera molt còmoda. És molt útil per guardar i manipular tota la informació dels models, però té un petit problema, i és la seva precisió en les operacions amb decimals. Per la gran majoria de càlculs no resulta un problema, però dins de l'algoritme de Quadric Errors hi ha un pas en el que s'ha d'invertir una matriu.

GLM té una funció per calcular inverses de manera automàtica, però el problema de calcular inverses bé quan el determinant de la matriu és proper a zero (la inversa es calcula com la adjunta entre el determinant d'una matriu). Com que estem tractant amb models molt complexos i detallats, els vèrtexs estan molt propers entre sí, i produeix matrius amb elements molts similars que fan que els determinants siguin molt petits i propers a zero, i per tant les inverses enormes, desembocant en una posició desmesurada. Aquí un parell d'exemples on els vèrtexs es disparen:





Per aquesta raó he necessitat utilitzar una altra llibreria. L'altra opció era implementar manualment jo mateix els càlculs d'inversa, però és millor aprofitar les llibreries que ja estan implementades i faciliten aquest tema. He decidit utilitzar la llibreria de **EIGEN**, que és molt eficient pels càlculs d'àlgebra lineal i té una gran precisió. La resta d'operacions les segueixo fent amb GLM, però per temes d'àlgebra lineal utilitzo les funcions de EIGEN. A més, també he creat una opció a l'aplicació gràfica perquè l'usuari pugui especificar un **llindar pel determinant**. Aquest llindar decideix si el determinant és massa petit per aplicar o no Quadric Errors en aquell vèrtex específic, important perquè no tots els models es comporten igual i alguns veurem que s'adapten diferent segons aquest llindar.

### 3.3.3 Rendiment

Per tancar el tema de Quadric Errors anem a analitzar quin cost afegeix a Vertex Clustering i Edge Collapse aquesta millora.

Comencem per **l'espai**, perquè aquesta vegada és el handicap més gran. Sigui n el nombre de vèrtexs del model original i m el nombre de triangles.

En el primer pas creem un vector de mida m, on cada posició és una matriu $K_p$ de 4x4 floats. Si suposem que un float són 4 bytes, aquest vector ocupa **64·m bytes.**

En el segon pas creem dos vectors, els dos de mida n. Del vector P(v), cada posició és una llista d'enters de mida constants (la quantitat de triangles incidents a un vèrtex sol ser de 4 de mitja), per tant direm que ocupa aproximadament **16·n bytes.** Del segon vector tenim que cada posició és una matriu $Q_s$ de 4x4 floats, per tant ocupa **64·n bytes.**

Fins aquí tenim lo que ocupa per Vertex Clustering, amb un total de **64·m + 80·n bytes.** Segueix sent lineal respecte la quantitat de geometria del model original, però ocupa molt més que tot Vertex Clustering sol. De fet, per al model del drac del principi que tenia 437.645 vèrtexs i 2.614.242 índexs, només amb les estructures utilitzades per Quadric Errors ocupa 91 milions de bytes (91 MB). Podem veure que l'espai és una de les principals limitacions de Quadric Errors, de fet en algunes execucions amb la meva aplicació s'ha aturat perquè el procés s'havia quedat sense memòria per assignar totes les matrius. Imaginat a més si afegim l'ampliació de Edge Collapse, que ha de guardar-se els errors de tots els vèrtex i anar actualitzant constantment aquesta info.

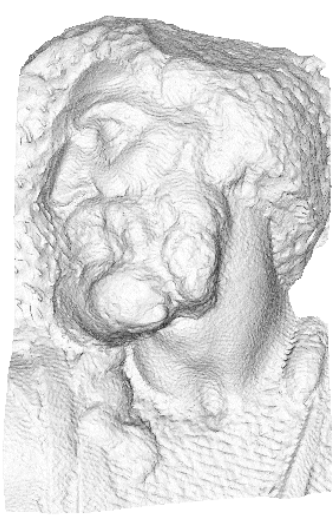
Pel que fa al cost temporal, no és tant crític degut a que són càlculs que es fan només un cop a l'inici per omplir totes les matrius. Primer hi ha un recorregut per tots els triangles, i després per tots els vèrtexs, amb un cost total de **θ(n + m).**

Cal notar també com ni el cost temporal ni espacial depenen del nombre de vèrtexs output, a diferencia dels altres algoritmes, així que serà un cost constant inicial indiferentment de la magnitud de simplificació que vulguis aconseguir.

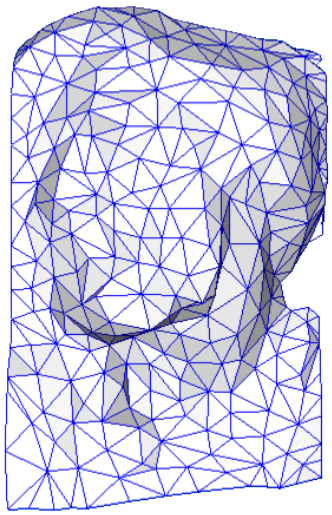
## 3.4 Millora Normal Mapping

Per finalitzar la secció de simplificació de models explicaré el quart algoritme que he investigat: **Normal Mapping**. De fet , aquesta no és una tècnica per reduir la geometria, sinó que més aviat s'utilitza un cop ja tens el model simplificat per augmentar el seu detall, sense augmentar la seva geometria.

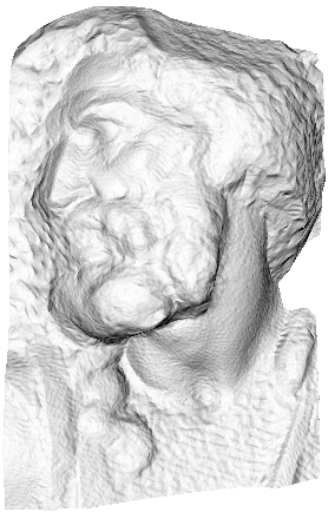
Al igual que Edge Collapse, degut a la falta de temps i que aquest algoritme és força complex i no és del tot necessari, no l'he implementat en la meva aplicació. Tot i així si que la comentaré, ja que forma part del procés ideal mencionat al principi, on primer reduïes la geometria del model i després intentaves que es veies lo més similar possible al model original afegint detall únicament a les textures. Explicació realitzada amb l'ajuda de la web **LearnOpenGL**.



Model original
4 milions de triangles

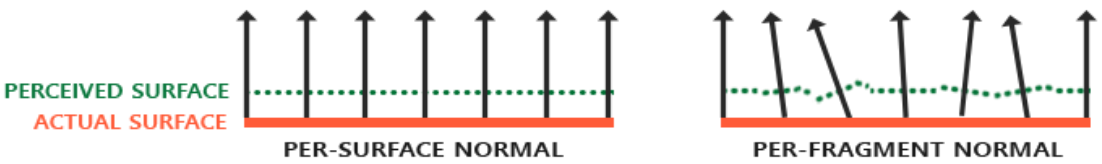


Model simplificat
500 triangles

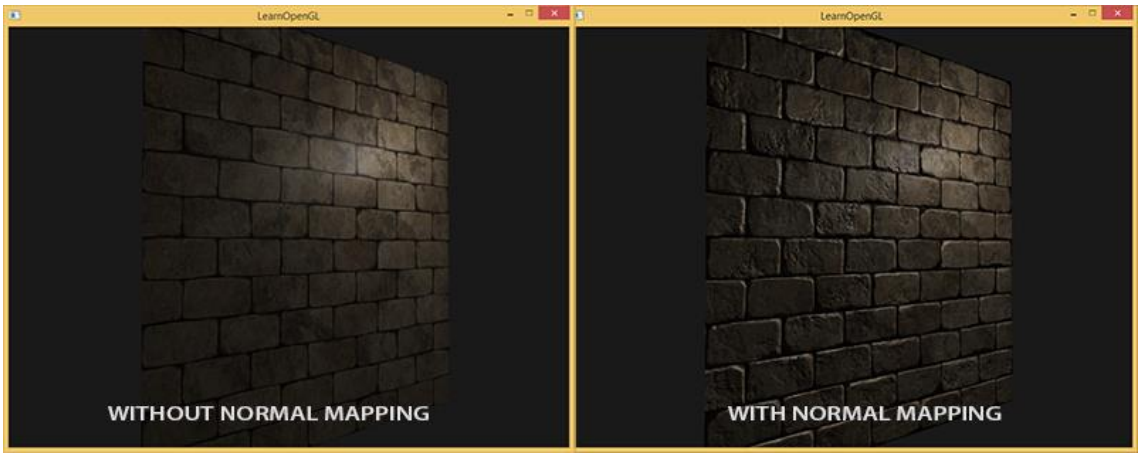


Model simplificat +
Normal Mapping
500 triangles

La millora de Normal Mapping es nota més en les superfícies rugoses que tenen petits detalls. En un model simplificat, el càlcul de la il·luminació no té en compte les petites esquerdes i forats. Es poden utilitzar mapes especulars per ajustar una mica la brillantor en certes zones, però al final la millor manera és informar directament als shaders de les normals que tenen els fragments, en comptes d'enviar-li normals per cara com teníem fins ara:



Aquest truc d'utilitzar normals per fragment en comptes de normals interpolades per cara es diu **Normal Mapping** i és una variació directa de **Bump Mapping**, el qual simula bumps i arrugues en les superfícies dels models pertorbant les seves normals. Si es realitza correctament (calculant la il·luminació adient en Tangent space, després veurem què és això) la millora és molt notòria:

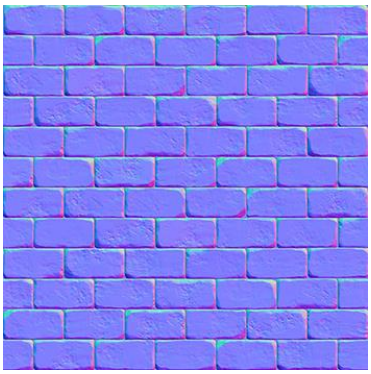



*Exemple de la diferència visual d'utilitzar Normal Map.*

### 3.4.1 Tècnica

L'algoritme es pot dividir en dos grans etapes: **obtenir el Normal Map del model original, i llegir les normals als shaders**.

Per tant, el primer que hem de fer és construir un Normal Map a partir del model original, col·locant a dins la informació de les normals de tots els vèrtexs. Els Normal Maps són textures 2D on cada píxel representa un color RGB. Acostumen a tenir un color blavós, degut a que en general les normals apunten cap a l'exterior en direcció de l'eix z positiu, representat amb el vector (0,0,1) que és la component blava RGB.



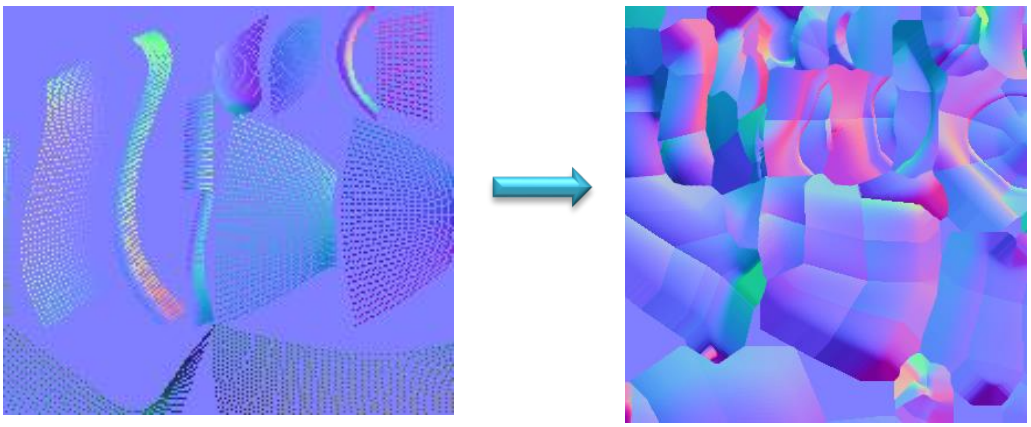
*Exemple de Normal Map*

Per passar un vector normal a un color per escriure'l en el Normal Map cal fer una petita transformació, ja que les coordenades de les normals tenen un rang entre [-1, 1] i els colors entre [0, 1]:

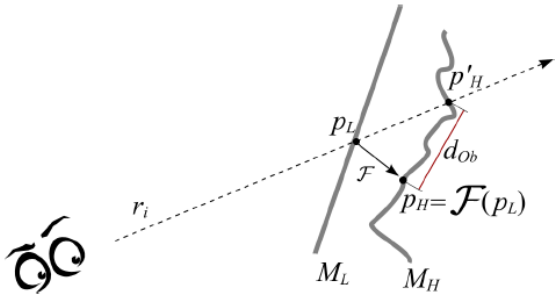
```
vec3 rgb_normal = normal * 0.5 + 0.5;
```

Un cop tenim una normal convertida en un color, sorgeix una nova pregunta. En quin píxel el col·loquem? Aquí sorgeix una altre discussió interesant. La solució és utilitzar les **coordenades de textura del vèrtex**. Això vol dir que el model ha de tenir coordenades de textura, per tant ha de venir amb alguna textura. Però què passa si és un model que encara no li han ficat cap material ni textura, com per exemple els de stanford? En aquest cas caldria generar coordenades manuals, i aquest és un procés extremadament complex de fer de manera eficient. Hi ha programes de modelatge, com Blender, que tenen la opció per fer lo que es coneix com **UV unwrap**, que consisteix en destripar la malla del model per assignar les coordenades, però insisteixo: és un procés difícil d'implementar, per aquesta raó he decidit no utilitzar-la en la meva aplicació.

A més, no tots els píxels de la textura tindran una coordenada assignada en algun vèrtex, per tant també cal interpolar els colors entre píxels. Això és complicat tenint en compte que existeixen discontinuïtats i salts dins de la textura:



Un altre mètode per generar aquests Normal Maps molt utilitzat és tenir en compte el model simplificat en el procés, i generar un Normal Map diferent per cada LOD. Aquest mètode està ben explicat en el paper de *Visibility based methods* [4], consisteix en utilitzar tècniques de **ray-casting** per tenir en compte els punts més propers entre el model original i el LOD, i calcular les respectives direccions. No cal indagar més en aquest assumpte, per està bé saber que existeixen diferents alternatives.



Suposant que ja tenim un Normal Map degudament generat, ara toca llegir la informació de les normals en els shaders per realitzar el càlcul correcte d'il·luminació.

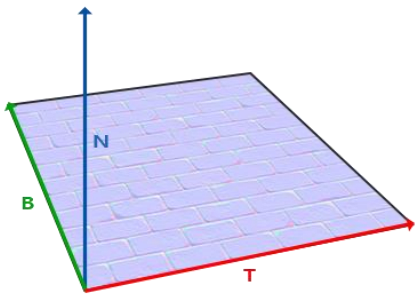
La lectura es fa de la mateixa manera que per els Diffuse Maps (les textures de color habituals). Donades les coordenades UV d'un fragment, es fa el mapejat en el fragment shader:

```
normal = texture(normalMap, texCoords).rgb;
normal = normalize(normal * 2.0 - 1.0);
```

Evidentment cal tornar a fer la transformació per passar del rang del color al rang de les normals. Un cop fet això ja tenim la nova normal, no cal tocar res més del shaders. El problema d'aquest senzill mètode és que aquestes normals estan expressades en un sistema de coordenades anomenat **Tangent space**, i els càlculs d'il·luminació es fan en el **Model space**, per tant no són del tot correctes.

El Tangent space és un sistema local al pla del triangle, on les normals són relatives a aquests plans. Cal fer un canvi de sistema, i en aquest punt ens seran útils els dos atributs dels vèrtexs que encara no havíem vist: **Tangents i Bitangents**.

A l'importar un model amb ASSIMP, una de les flags que li podem passar és la de **aiProcess_CalcTangentSpace**, que genera de manera automàtica aquests dos atributs (sempre i quan el model tingui TexCoords, en aquest procés també és obligatori que el model vingui amb alguna textura assignada). De totes maneres les tangents i bitagents també es poden calcular manualment. Cal tenir en compte que una Normal, una Tangent i una Bitangent han de ser ortogonals entre sí.




Un cop tenim una normal, una tangent i una bitangent per cada vèrtex, en el vèrtex shader calculem la matriu de canvi de sistema anomenada TBN:

```
vec3 T = normalize(vec3(model * vec4(aTangent,   0.0)));
vec3 B = normalize(vec3(model * vec4(aBitangent, 0.0)));
vec3 N = normalize(vec3(model * vec4(aNormal,    0.0)));
mat3 TBN = mat3(T, B, N);
```

I la passem al fragment shader, on l'utilitzem per posar la normal extreta del Normal Map com toca:

```
normal = normalize(TBN * normal);
```

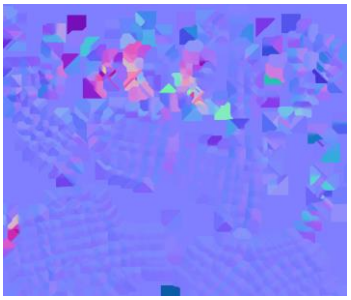
### 3.4.2 xNormal

Per tancar aquest tema, comentar que per descomptat existeixen programes externs que intenten generar Normal Maps, tot i que a vegades no acaben de funcionar bé.

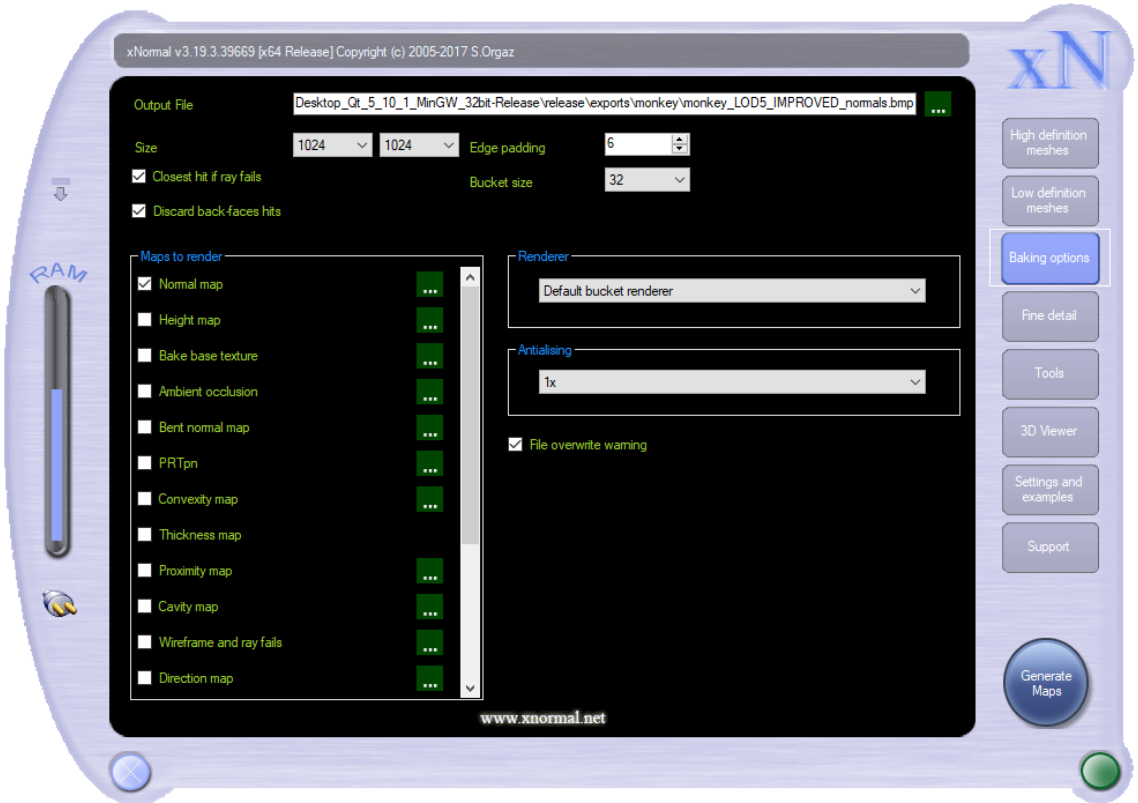
Un exemple que he provat és xNormal, que utilitza el mètode de generar un Normal Map tenint en compte el model simplificat. Es pot descarregar des de la seva web: http://www.xnormal.net/ .

És fàcil i ràpid d'utilitzar, i un punt fort és que es pot cridar amb una comanda i per tant la podria utilitzar des de dins de la meva aplicació. Tot i així, després de provar-la també presenta molts problemes, i al final m'he decantat per no fer-la servir.

Alguns dels problemes són:



- Moltes textures apareixen corruptes, potser problema a l'hora de calcular les normals, o les tangents i bitangents.
- No pot executar-se si el model no té coordenades de textura, tal i com he comentat abans.
- Només tracta amb models que tenen un únic mesh, però la majoria dels models sempre estaran composats per múltiples meshes (el tractament seria enviar cada mesh per separat però llavors cal exportar prèviament cada mesh, procés feixuc).




*Interfície de xNormal*

# Aplicació Gràfica

Ara que he explicat totes les tècniques que he investigat m'agradaria dedicar aquesta secció a ensenyar la **aplicació gràfica** que he construït, amb la implementació completa de **Vertex Clustering** i **Quadric Errors**. Repassaré totes les funcionalitats i opcions disponibles, com s'utilitza i com s'obté.


L'he anomenat **SceneOptimizer**, que en anglès vol dir "Optimitzador d'Escenes". L'aplicació sencera està escrita en anglès, tant la interfície com la informació disponible (perquè així és més accessible per a tothom que la vulgui utilitzar). També he dissenyat una icona simple i elegant que mostra les lletres pixelades SO del nom.

El codi l'he desenvolupat en **QT Creator 4.6.0** i la interfície gràfica en **QT Designer** (inclòs en QT Creator). La versió de QT és **QT 5.10.1**. El projecte està compilat en **Release** i amb el compilador **MinGW 32bit.**

## 4.1 Descàrrega

**SceneOptimizer** és un executable .exe i ve acompanyat per altres carpetes i elements que després veurem. La carpeta sencera pesa 104 MB (comprimida pesa 23 MB). **Es pot descarregar des d'aquest enllaç de MEGA:**

https://mega.nz/#!IkQS1YqT!dVyDC1qzLTngjga3AhIXZ7FlRPIqPGWSnEJYJDmqZK8

Addicionalment també he penjat la carpeta que conté tots els **models** de prova que he utilitzat. No l'he inclòs a la carpeta de SceneOptimizer perquè pesa bastant, 898 MB (comprimida pesa 228 MB). **Es pot descarregar des d'aquest enllaç de MEGA:**

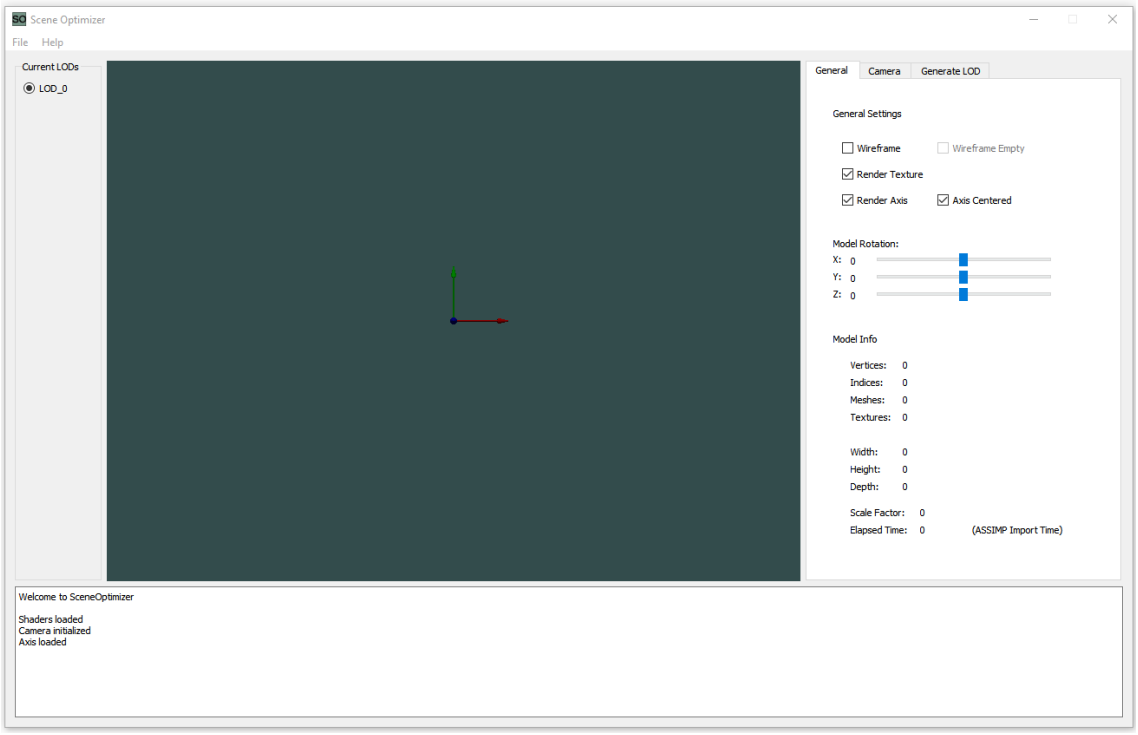
https://mega.nz/#!lgBnWCoY!coJhIiTCkd0uQvyPW9T3Si6E7dDCfpackn6qUlce7j8

El **codi font** també l'he posat disponible. La carpeta sencera pesa 105 MB (8 MB comprimida). **Es pot descarregar des d'aquest enllaç de MEGA:**

https://mega.nz/#!BhRHzYIT!HhFDSP6ir6R-_ioNhq4uRl1i2sGIjBxJ5QFGyEfLVMc

## 4.2 Descripció de funcionalitats

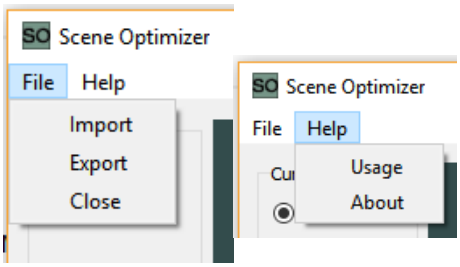
Un cop executes SceneOptimizer et surt la **pantalla principal (Main Window)**:




Aquesta pantalla està formada per un menú i uns quants widgets. En el menú situat a dalt a la esquerra hi ha dos desplegables:

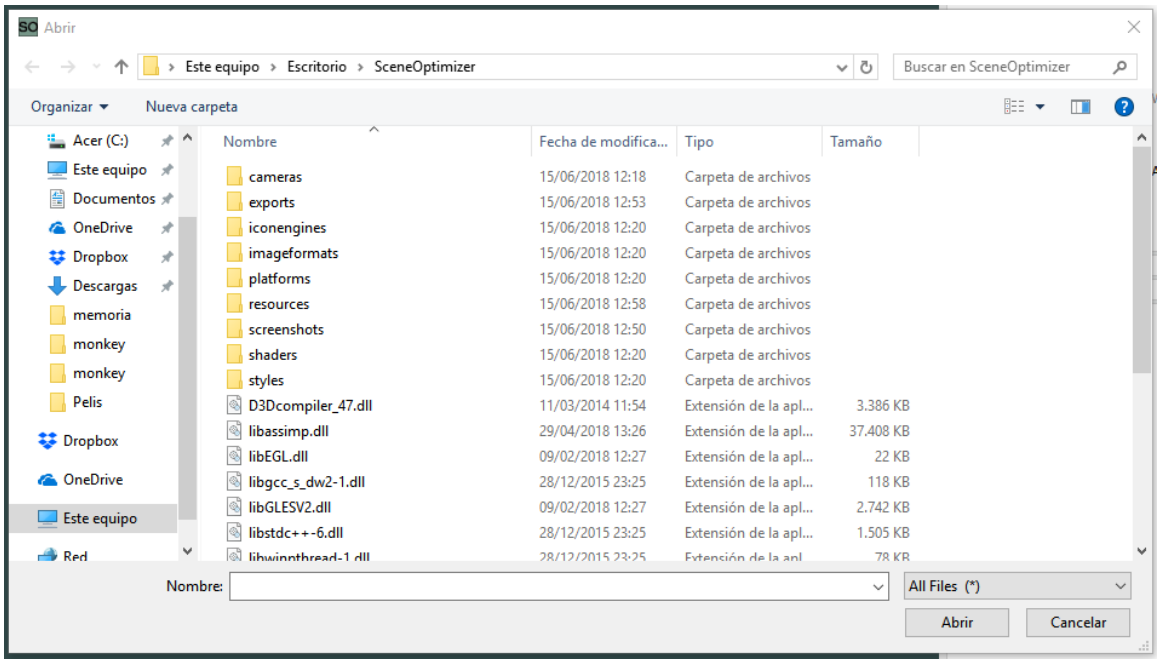
**File,** que conté les opcions per importar i exportar un model i tancar l'aplicació.

**Help,** que conté les opcions per veure el Usage i l'About.

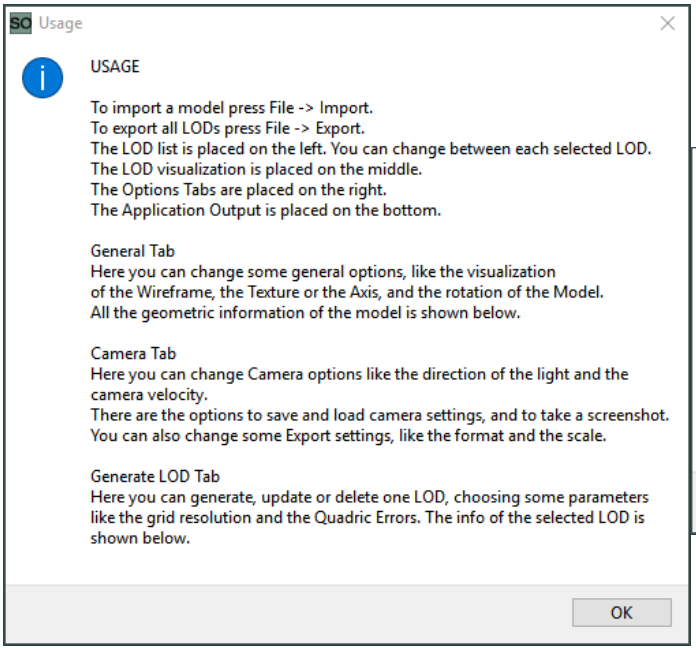



Tant per **importar** com per **exportar** s'obra un File Browser, que et permet navegar pel sistema de carpetes per seleccionar l'arxiu que vulguis.

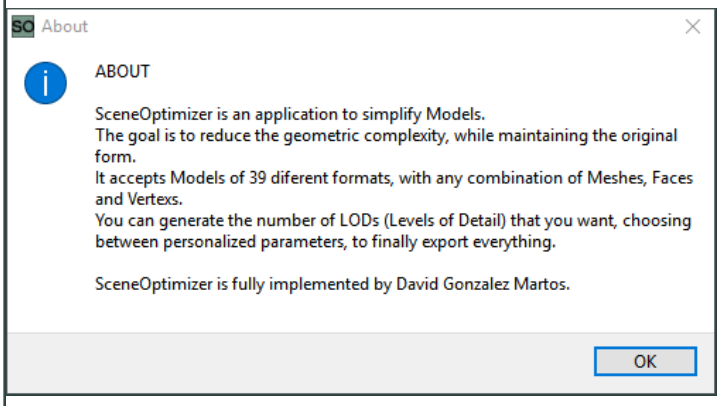
**Usage** i **About** obren un Dialog informatiu amb un text escrit, en el primer cas sobre l'ús de l'aplicació i en l'altre sobre informació general.




*File Browser per cercar un arxiu*




*Finestra Usage*



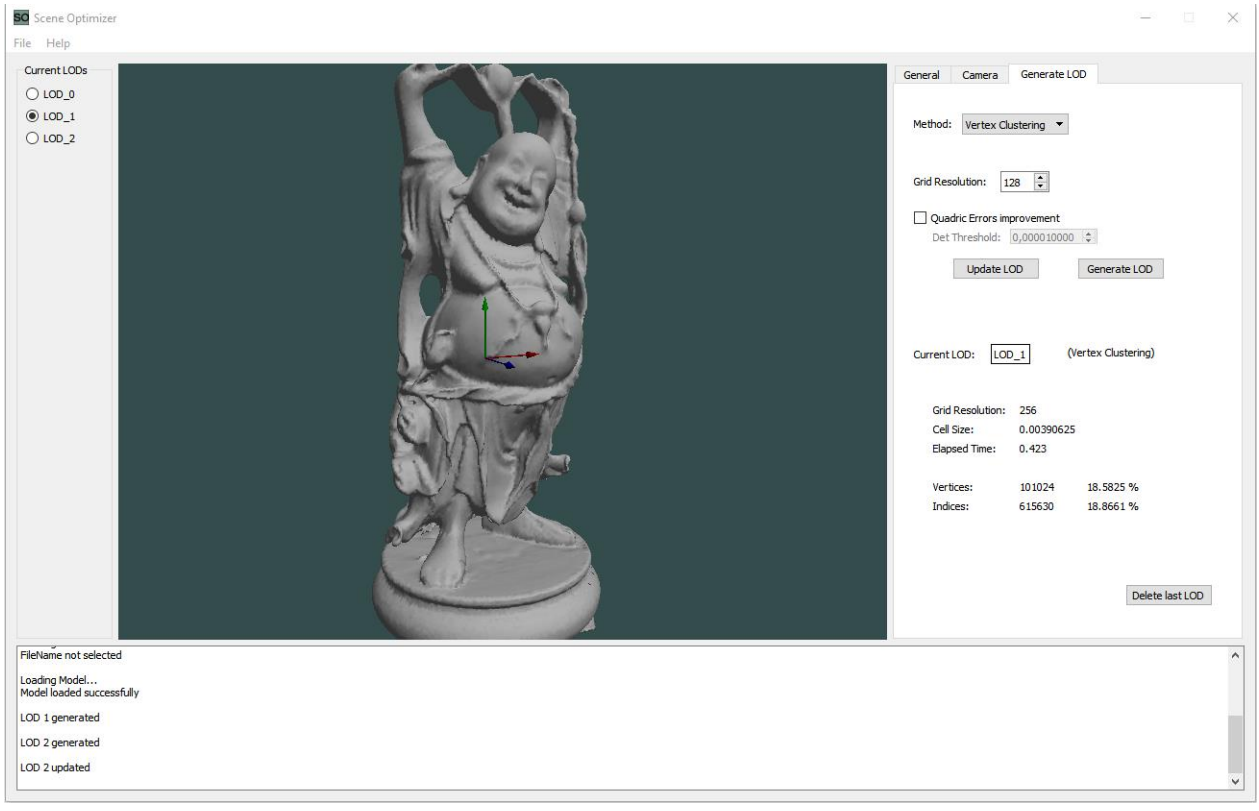

*Finestra About*

A la part esquerra de la finestra principal està situat la **llista de LODs**, on es mostraran tots els LODs generats per al model actual. Es pot canviar entre cada LOD per veure el resultat visual i la seva informació.

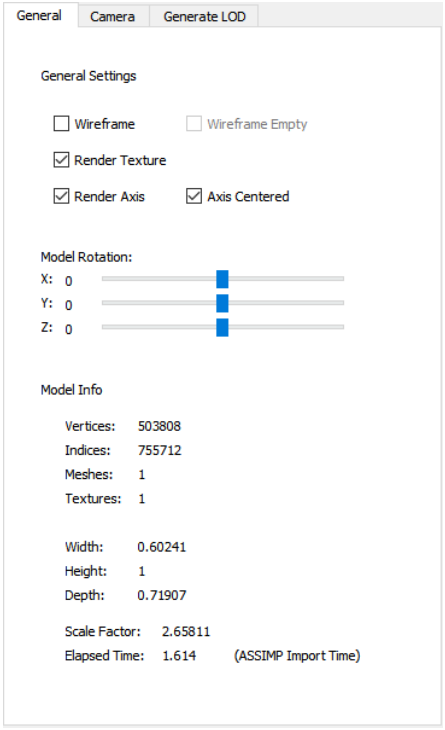
La pantalla del mig és on es **renderitza l'escena** amb el model importat i amb les característiques explicades al principi del document (Il·luminació i shaders, càmera interactiva que et permet moure de manera lliure, etc). Només es renderitza el LOD seleccionat, sent el LOD 0 el model original. A més, també he creat uns **eixos de coordenades** renderitzats en l'origen de coordenades, sent l'eix x vermell, l'eix y verd i l'eix z blau. Aquests eixos faciliten veure on està situat el centre de l'escena, i les direccions del model.

A la dreta tenim el panell amb totes les opcions. Hi ha 3 pestanyes: **General**, **Camera** i **Generate LOD**, que ara explicaré amb detall.

A la part de baix hi ha un quadre de text per mostrar **l'output de l'aplicació**, on apareixen totes les accions que ocorren durant l'execució de l'aplicació.

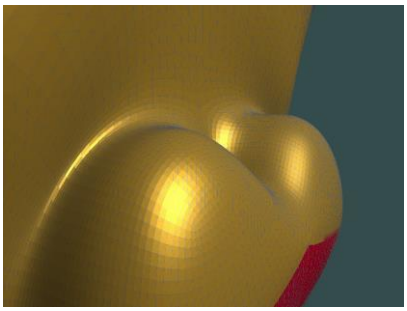


### 4.2.1 Pestanya General

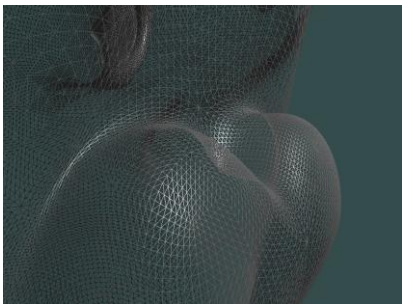



Aquí hi ha les opcions generals del model importat.

Per començar, he implementat una funció per renderitzar el **Wireframe** del model, que és el filferro resultant quan uneixes tots els vèrtexs amb línies, sense dibuixar cares. És molt útil si vols veure com és la geometria. Hi ha un *checkbox* per renderitzar el Wireframe per sobre del model, i un altre sense el model.



*Wireframe sobre el model*



*Wireframe sense el model*

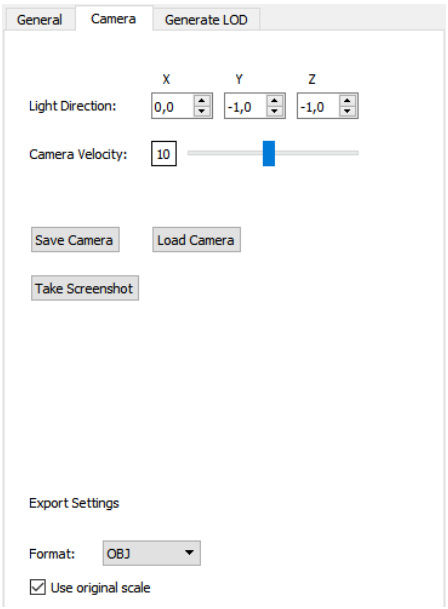
Seguidament hi ha un *checkbox* per renderitzar o no la **textura** del model. En cas de seleccionar que no, o si el model directament no porta textura, el programa utilitza una textura per defecte de dimensions 1x1 que he creat, de color gris. Útil si et vols centrar en la geometria del model i ho vols veure tot del mateix color.

Després tenim dos checkboxs més, un per **renderitzar els eixos** que he mostrat abans, i un altre per indicar-li si vols que estiguin a **l'origen de coordenades** o a l'origen del model. Útil per si et molesta i el vols treure.

A continuació venen tres *sliders* per determinar **la rotació del model**. En general no caldrà tocar aquesta opció, ja que és més còmode moure's per l'escena amb la càmera i posicionar-te on vulguis, però poden haver ocasions on els models apareguin completament girats per culpa del canvi de sistema amb altres programes (la regla de la ma esquerra, etc.) En aquests casos és més fàcil rotar directament el model en l'eix que vulguis.

Per últim es mostra la **informació geomètrica del model importat**, per saber els vèrtexs, índexs, cares, meshes i textures que conté. Es mostren les seves dimensions (recordem, escalades a l'inici per que tot s'ajusti a la escena) i finalment el factor d'escalat i el temps que ha trigat ASSIMP per importar el model sencer, que sol ser el coll d'ampolla i la part que triga més de tot el programa. El temps es mostra en segons.

### 4.2.2 Pestanya Camera




Aquí tenim les opcions relacionades amb la càmera i la il·luminació.

Per començar podem modificar la **direcció de la llum** de l'escena en qualsevol dels eixos. L'interval és entre [-1,1], i si poses 0 en totes direccions és equivalent a eliminar la llum.

També hi ha un *slider* per adaptar **la velocitat de la càmera** segons estigui còmode l'usuari. Aprofito per recordar que prement la tecla Ctrl redueixes la velocitat, i amb la tecla Shift la augmentes.

A continuació tenim tres botons que faciliten la visualització al canviar entre models. Com que cada vegada que s'importa un model es reinicien totes les opcions, aquestes són opcions són útils si vols carregar diferents models i veure'ls tots desde la mateixa perspectiva.

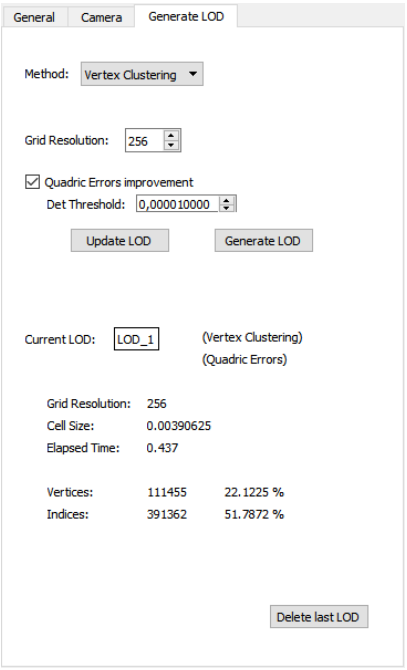
Amb el botó **Save Camera** guardes la configuració actual de la càmera. Aquesta informació és guarda per defecte en la carpeta "cameras", en un txt amb el nom que especifiquis.

Amb el botó **Load Camera** carregues una de les configuracions guardades.

Amb el botó **Take Screenshot** guardes una captura de pantalla del model. Aquesta imatge és un png que per defecte es guarda a la carpeta "screenshots".

A la part de sota també he inclòs un parell d'opcions per **exportar** el contingut. Quan s'exporta, es guarden tots els LODs generats per al model actual, generant els arxius necessaris automàticament i copiant les textures utilitzades. Aquí pots escollir el format en el que vols exportar i si vols utilitzar l'escala original o la actual. De moment només hi ha disponibles tres formats: **OBJ** (Wavefront Object), **DAE** (Collada) i **PLY** (Stanford Triangle Format). Aquests són per sort els més utilitzats, però és una llàstima que la llibreria ASSIMP tingui implementats tant pocs formats per exportar quan per importar en té 39.

### 4.2.3 Pestanya Generate LOD




Aquí es troben les opcions relacionades amb la simplificació i la generació de LODs.

Per començar hi ha un desplegable que et permet triar el **mètode de simplificació**, però de moment només està disponible Vertex Clustering (he posat el desplegable per si amplio aquesta aplicació en el futur i implemento també Edge Collapse).

Seguidament es pot triar les **resolució de la graella**. És el nombre de cel·les en les tres direccions que tindrà la graella de Vèrtex Clustering, podent escollir entre 1 i 2048. Per defecte està en 256, que en general produeix bons resultats. Si el mètode triat fos Edge Collapse, en comptes de la resolució es podria escollir el nombre de triangles output que volguessis.

També hi ha un checkbox per dir si vols utilitzar **la millora de Quadric Errors** o no, que recordem que millora les posicions dels nous vèrtexs generats. Aquesta opció estaria disponible també per Edge Collapse. A més pots especificar un **llindar pel determinant**, ja que per determinants molt baixos hem comentat que hi ha problemes de precisió.

Seguidament hi ha dos botons, un per **generar un LOD nou** i l'altre per **actualitzar el LOD actual** seleccionat a la llista de l'esquerra. Es tenen en compte els valors anteriors especificats.

A sota hi ha un botó per **eliminar l'últim LOD** creat. Faig que sigui sempre l'últim perquè no té sentit que els LODs estiguin desordenats.

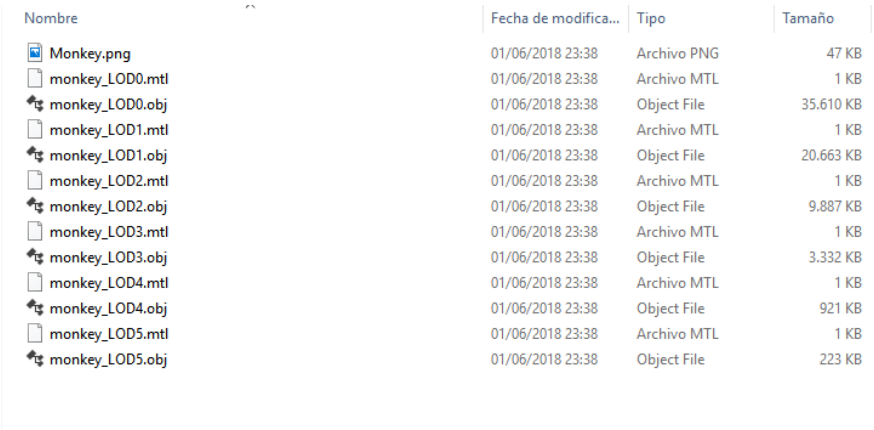
I per acabar es mostra la **informació geomètrica del LOD actual** seleccionat. Et mostra el nom del LOD, el mètode utilitzat per simplificar-lo i els paràmetres escollits, així com els vèrtexs i índexs output. També es mostra el percentatge de geometria reduïda respecte el model original.

## 4.3 Altres continguts de l'aplicació

Quan et descarregues l'aplicació, SceneOptimizer ve dins d'una carpeta amb altres arxius i carpetes. La gran majoria d'arxius són **dlls** necessaris per la correcta execució, afegits per el mateix QT Creator després de realitzar un **Deployment** de l'aplicació. Les carpetes més importants són:

**cameras**. Aquí es guarden per defecte les configuracions de les càmeres guardades, en fitxers .txt que també poden ser modificats.

**exports**. Aquí es guarden per defecte els LODs quan s'exporten. Quan prems exportar, el File Browser et permet triar un nom comú per totes les simplificacions d'un model, però després cada LOD es numera adequadament amb el sufix "_LODi":



| Nombre | Fecha de modifica... | Tipo | Tamaño |
|---|---|---|---|
| Monkey.png | 01/06/2018 23:38 | Archivo PNG | 47 KB |
| monkey_LOD0.mtl | 01/06/2018 23:38 | Archivo MTL | 1 KB |
| monkey_LOD0.obj | 01/06/2018 23:38 | Object File | 35.610 KB |
| monkey_LOD1.mtl | 01/06/2018 23:38 | Archivo MTL | 1 KB |
| monkey_LOD1.obj | 01/06/2018 23:38 | Object File | 20.663 KB |
| monkey_LOD2.mtl | 01/06/2018 23:38 | Archivo MTL | 1 KB |
| monkey_LOD2.obj | 01/06/2018 23:38 | Object File | 9.887 KB |
| monkey_LOD3.mtl | 01/06/2018 23:38 | Archivo MTL | 1 KB |
| monkey_LOD3.obj | 01/06/2018 23:38 | Object File | 3.332 KB |
| monkey_LOD4.mtl | 01/06/2018 23:38 | Archivo MTL | 1 KB |
| monkey_LOD4.obj | 01/06/2018 23:38 | Object File | 921 KB |
| monkey_LOD5.mtl | 01/06/2018 23:38 | Archivo MTL | 1 KB |
| monkey_LOD5.obj | 01/06/2018 23:38 | Object File | 223 KB |

**resources**. Aquí es troba el model dels eixos situats a l'origen de coordenades. També està la textura per defecte que s'aplica als models sense textura (o quan selecciones no renderitzar textura) i la icona de la aplicació. Inicialment també estaven tots els models de prova, però al final els he tret ja que ocupaven massa espai (casi 1 GB) i no aportaven res a la aplicació en si. Únicament he deixat el model del Monkey, com sample model.

**screenshots**. Aquí es guarden per defecte les captures de pantalla. Pots triar el nom que vulguis, però sempre es guarden en el format .png.

**shaders**. Aquí es troben els dos shaders bàsics que utilitzo per al correcte renderitzat i càlcul de la il·luminació de l'escena. Hi ha un vèrtex shader i un fragment shader.

# Proves i Resultats

Arribats a aquest punt, ja he explicat la part teòrica i algorítmica de les tècniques investigades, a més de mostrar l'aplicació gràfica que he desenvolupat per utilitzar les tècniques que he decidit implementar. Ara falta veure **què puc obtenir amb SceneOptimizer**, falta provar amb models reals de diferents complexitats i característiques per veure els resultats de la seva simplificació, i si realment tot funciona de manera correcta i esperable.

Durant la primera part mostraré els resultats obtinguts amb tots els models de prova que tinc, comentant detalls i comparant amb diferents paràmetres de simplificació.

Després utilitzaré alguns dels LODs obtinguts a la primera part per carregar-los a **Unity**, un motor de videojocs, per veure realment la millora de rendiment a la pràctica, i provant de canviar entre diferents LODs en directe (mentre el joc s'està executant).

Finalment faré una reflexió del que he aconseguit durant aquest projecte i el possible treball futur que quedaria per acabar d'ampliar i millorar l'aplicació gràfica.

El procés que seguiré per fer les proves és el següent:

1. Per cada model, mostraré una imatge sencera del **model original** i especificaré les seves característiques més rellevants (vèrtexs, triangles, etc.).
2. Col·locaré tres imatges corresponents a tres simplificacions generades amb **Vertex Clustering**, i a sota exactament les mateixes simplificacions però aquest cop amb **Vertex Clustering + Quadric Errors**.
3. Crearé una **taula** per especificar els resultats geomètrics de cada versió. Com que les simplificacions seran les mateixes, l'únic que canviarà entre cada una és el temps expressat en segons (i el resultat visual).
4. Finalment **interpretaré** aquestes dades i faré un **anàlisi** visual de com es veuen els resultats, que realment és lo més important.

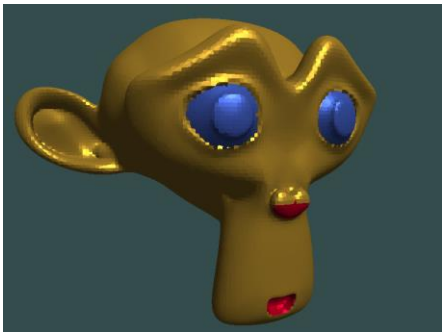
No utilitzaré tots els models disponibles, sinó només aquells que presentin resultats interessants (tant bons com dolents).

## 5.1 Monkey

| | |
|---|---|
| Vèrtexs: | 503,808 |
| Índexs: | 755,712 |
| Triangles: | 251,904 |
| Meshes: | 1 |
| Té textura: | Sí |

**Vertex Clustering:**

*LOD 1*

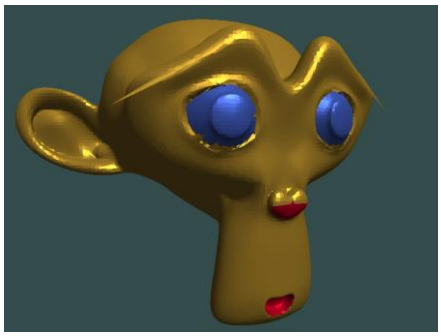


*LOD 2*

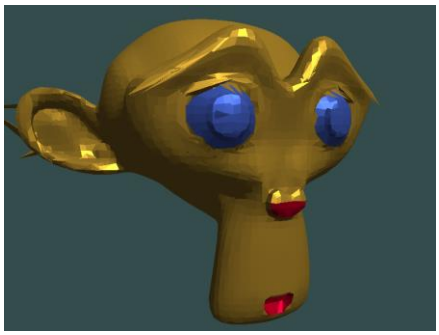
*LOD 3*

**VC + Quadric Errors:**

*LOD 1*



*LOD 2*



*LOD 3*

**Resultats** (el temps és en segons) (VC = Vertex Clustering) (QE = VC + Quadric Errors):

| | Grid Resolution | VC Time | QE Time | Vertexs | Indexs |
|---|---|---|---|---|---|
| **LOD 1** | 256 | 0.31 | 0.36 | 111,455 (22 %) | 391,362 (52 %) |
| **LOD 2** | 128 | 0.26 | 0.3 | 70,767 (14 %) | 164,412 (22 %) |
| **LOD 3** | 64 | 0.21 | 0.25 | 30,245 (6 %) | 49,392 (7 %) |

**Anàlisi:**

Comencem amb el model que més he utilitzat. Conté textura, per tant també serveix per veure si els atributs es calculen correctament (sobre tot les coordenades de textura, que visualment criden molt l'atenció si estan malament).

Pel que fa al **temps de càlcul, és molt reduït** en els dos casos. Amb Quadric Errors tal i com esperàvem té un petit cost addicional al principi, però no suposa pas un gran afegit.

La **taxa de reducció de dades és molt elevada**, de fet només amb el primer LOD ja es redueix casi el 80% dels vèrtexs, en comparació amb el 50% dels índexs (triangles). Aquesta diferència entre vèrtexs i índexs indica que el model conté molts vèrtexs repetits, és a dir, que tenen la mateixa posició però diferents atributs. Això no és incorrecte, ni vol dir que estigui mal modelat, senzillament es fa per mantenir les discontinuïtats de normals, tex coords, etc. Això provoca que encara que augmenti la resolució de la graella per intentar simplificar menys se seguiran eliminant una gran quantitat de vèrtexs, tots els que estiguin repetits.

Pel que fa a Vertex Clustering el model es simplifica sense problemes, donant un **aspecte més pixelat** a mesura que es redueix més. Els **atributs es preserven molt bé**, de fet les coordenades de textura estan perfectes i les normals són bastant estables també. La forma també es manté intacta, no existeix cap deformitat ni apareix cap artifact. Podem concloure per aquest model que **els resultats són molt satisfactoris amb Vertex Clustering**.

Per Quadric Errors no podem dir el mateix. Com aquesta millora només afecta a les posicions dels vèrtexs, la resta d'atributs es segueixen mantenint correctament. A mesura que es simplifica més, es poden apreciar que alguns vèrtexs es disparen i acaben en **una posició totalment incorrecte**, sobretot al voltant dels ulls i les orelles. He provat amb diferents llindars pel determinant, però no soluciona pràcticament res, per tant no és problema de errors de precisió. Podem concloure que **la millora de Quadric Errors no produeix bons resultats** per aquest model en concret.

## 5.2 Nanosuit

| | |
|---|---|
| Vèrtexs: | 57,174 |
| Índexs: | 57,174 |
| Triangles: | 19,058 |
| Meshes: | 7 |
| Té textura: | Sí |

**Vertex Clustering:**

*LOD 1*

*LOD 2*

*LOD 3*

**VC + Quadric Errors:**

*LOD 1*

*LOD 2*

*LOD 3*

**Resultats:**

| | Grid Resolution | VC Time | QE Time | Vertexs | Indexs |
|---|---|---|---|---|---|
| **LOD 1** | 256 | 0.033 | 0.037 | 11,370 (20 %) | 45,906 (80 %) |
| **LOD 2** | 128 | 0.029 | 0.033 | 9,690 (17 %) | 34,002 (59 %) |
| **LOD 3** | 64 | 0.025 | 0.029 | 6,927 (12 %) | 18,993 (33 %) |

**Anàlisi:**

Aquest cop he provat amb un model que ha donat resultats millors per Quadric Errors. És un amb menor geometria, i es nota molt en el temps, que **és pràcticament instantani** en els dos casos.

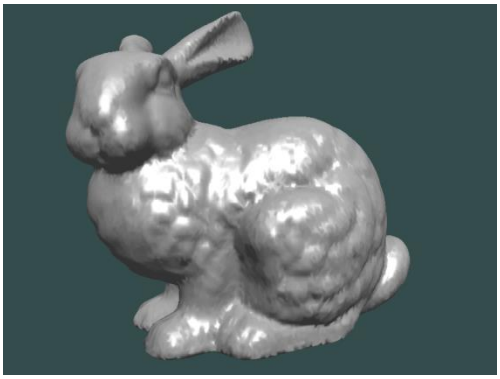
En aquest **la taxa de reducció de dades és encara major** per als vèrtexs, reduint exactament el 80% dels originals. En canvi per als triangles li costa més, s'ha de simplificar bastant per arribar a eliminar dos terços dels índexs originals.

Aquest model també té textura, i a més està compost per diversos meshes, a diferencia del monkey. **Els atributs es preserven de manera molt correcta** en els dos casos.

Ara la gran diferencia la tenim amb les posicions dels vèrtexs. És possible que tal i com estan posades les imatges no es noti gaire, però si es posen una sobre l'altra i es canvien com si fos un GIF, s'apreciarà clarament que **Quadric Errors preserva millor la forma**. En cap cas es produeixen posicions errònies, però Vertex Clustering tendeix a fer més petit aquests tipus de models, ja que les posicions mitjanades es van cap en dins, com si el model fos una mica més prim. En canvi Quadric Errors aconsegueix mantenir més correctament el volum real del model. Només en els casos més extrems de simplificació es perden detalls vistosos, però en els dos casos, com per exemple en el tercer LOD que el dit polze de la ma dreta casi desapareix.

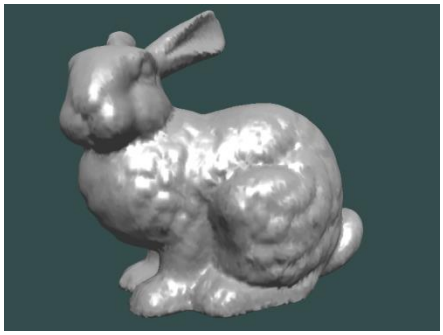
També és un model interessant per comentar **el problema de la connectivitat entre diversos meshes**. La simplificació es fa individual per cada mesh, per tant és possible que es perdin les connexions que pugui tenir el model original. Aquest problema és totalment esperable, i casi ni es nota, només en casos molt extrems. Un exemple on es nota això si t'apropes molt és en el canell de la ma dreta del nanosuit, que poc a poc es va separant.

## 5.3 Bunny

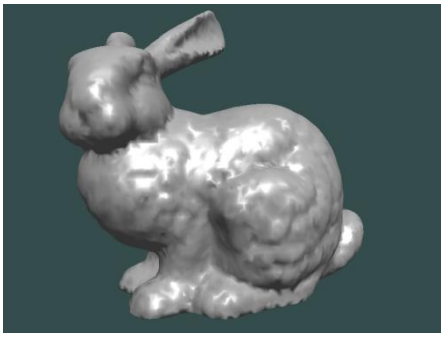


| | |
|---|---|
| Vèrtexs: | 35,947 |
| Índexs: | 208,353 |
| Triangles: | 69,451 |
| Meshes: | 1 |
| Té textura: | No |

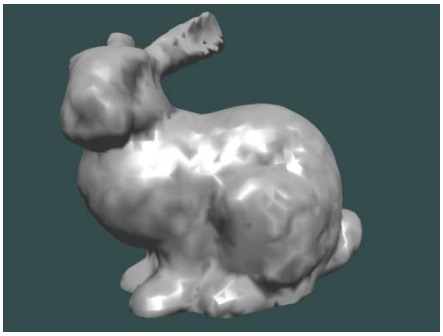
**Vertex Clustering:**



*LOD 1*

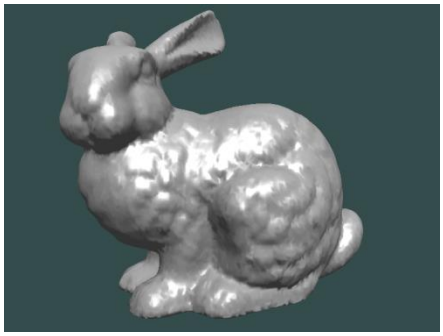


*LOD 2*

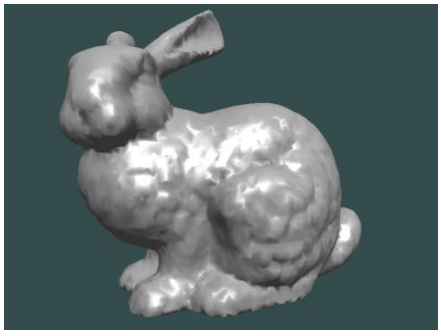


*LOD 3*

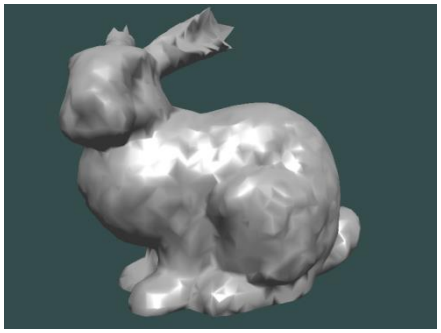
**VC + Quadric Errors:**



*LOD 1*



*LOD 2*



*LOD 3*

**Resultats:**

| | Grid Resolution | VC Time | QE Time | Vertexs | Indexs |
|---|---|---|---|---|---|
| **LOD 1** | 128 | 0.038 | 0.088 | 30,498 (85 %) | 180,045 (86 %) |
| **LOD 2** | 64 | 0.029 | 0.061 | 11,279 (31 %) | 67,239 (32 %) |
| **LOD 3** | 32 | 0.025 | 0.050 | 3,138 (9 %) | 18,891 (9 %) |

**Anàlisi:**

Ara toca veure un cas on el model no té textura, com per exemple els models descarregats de la web de stanford.

El bunny és un model geomètricament simple, per tant **els temps de càlcul són mínims**. Tot i així, sembla que l'offset inicial de Quadric Errors és una mica superior en comparació als dos casos anteriors, de fet està duplicant el temps. Té molt de sentit, perquè el temps de Veretx Clustering està directament relacionat amb els vèrtexs output, i Quadric Errors en canvi és lineal en relació a la geometria del model original. Aquest model té pocs vèrtexs però molts índexs en comparació, per tant **el temps afegit és culpa dels triangles**.

Aquest cop es nota que el model no necessita preservar coordenades de textura, i que per tant no conté vèrtexs repetits. **La taxa de reducció és molt similar** tant per vèrtexs com per índexs, i cau de manera proporcional a mesura que es divideix la resolució de la graella.

Per als dos casos es redueix de manera bastant correcta, però per Quadric Errors sembla que al final alguns vèrtexs de la orella acaben en posicions una mica desplaçades. Realment tot apunta a que **Quadric Errors funciona pitjor quan major és la simplificació**.

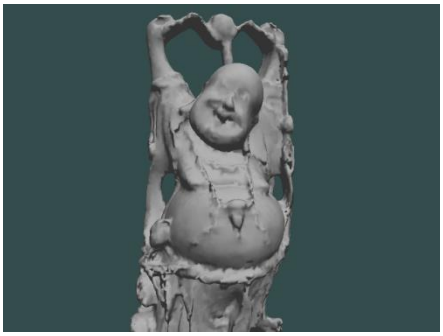
## 5.4 Happy Buda

| | |
|---|---|
| Vèrtexs: | 543,652 |
| Índexs: | 3,263,148 |
| Triangles: | 1,087,716 |
| Meshes: | 1 |
| Té textura: | No |

**Vertex Clustering:**

*LOD 1*

*LOD 2*



*LOD 3*

**VC + Quadric Errors:**

*LOD 1*

*LOD 2*

*LOD 3*

**Resultats:**

| | Grid Resolution | VC Time | QE Time | Vertexs | Indexs |
|---|---|---|---|---|---|
| **LOD 1** | 512 | 0.58 | 1.08 | 260,575 (48 %) | 1,568,490 (48 %) |
| **LOD 2** | 256 | 0.48 | 0.75 | 101,024 (19 %) | 615,630 (19 %) |
| **LOD 3** | 128 | 0.33 | 0.57 | 29,842 (5 %) | 185,634 (6 %) |

**Anàlisi:**

Fins el moment em vist models amb una quantitat de geometria moderada. Aquest és força més complex, amb més de 3 milions de índexs i mig milió de vèrtexs.

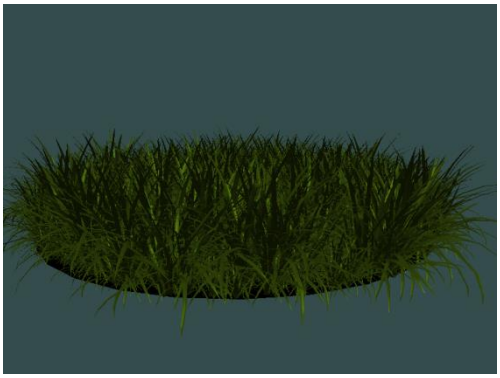
Tot i així **el temps d'execució segueix sent increïblement ràpid**, arribant a penes al segon amb Quadric Error i resolució elevada.

La **taxa de reducció de dades és simètrica** per vèrtexs i índexs, similar amb el Bunny.

Una component interessant d'aquest model és **la quantitat de detall**. Al voltant del seu cos té diversos elements amb un detall molt elevat, i sorprenentment **es preserva molt bé** al llarg de les simplificacions. Només en el LOD 3 es comencen a veure una mica malament algunes parts, però cal recalcar que en aquest LOD s'han eliminat ja el 95 % dels vèrtexs, per tant és increïble que encara és mantingui tant bé la forma.

Per aquest model **Quadric Errors no té diferències significatives**, visualment no sembla que millori ni empitjori res. Per alguns models es nota més que en d'altres.

## 5.5 Grass

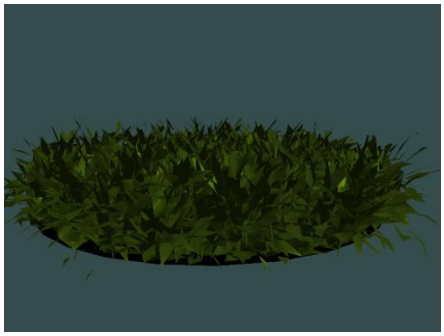


| | |
|---|---|
| Vèrtexs: | 1,565,056 |
| Índexs: | 2,271,858 |
| Triangles: | 757,286 |
| Meshes: | 3 |
| Té textura: | Sí |

**Vertex Clustering:**

*LOD 1*

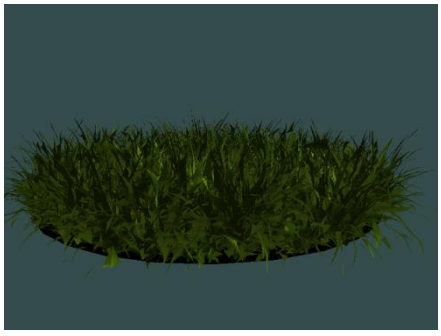
*LOD 2*



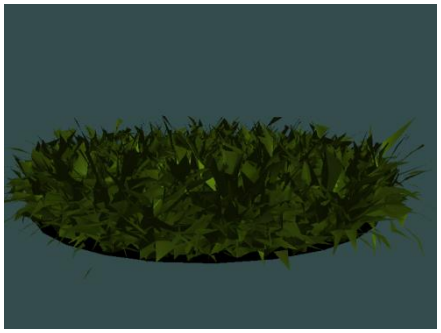
*LOD 3*

**VC + Quadric Errors:**

*LOD 1*



*LOD 2*



*LOD 3*

**Resultats:**

| | Grid Resolution | VC Time | QE Time | Vertexs | Indexs |
|---|---|---|---|---|---|
| **LOD 1** | 256 | 1.04 | 1.40 | 302,261 (19 %) | 783,174 (34 %) |
| **LOD 2** | 128 | 0.82 | 0.96 | 150,273 (10 %) | 360,285 (16 %) |
| **LOD 3** | 64 | 0.67 | 0.79 | 53,015 (3 %) | 138,948 (6 %) |

**Anàlisi:**

Per acabar m'agradaria investigar una mica més la preservació del detall, sobretot tenint en compte que per Vertex Clustering hauria de ser complicat però tot i així estic obtenint resultats sorprenentment satisfactoris. Per fer això he triat un model que conté **detall en extrem**: un tros d'herba. Com que cada trosset està modelat, conté moltíssima geometria amb 1 milió i mig de vèrtexs (el model més complex que hem vist fins ara).

El temps segueix estan **al voltant del segon**, i **la taxa de reducció de dades és enorme** en el primer LOD, indicant que efectivament hi ha un gran nombre de vèrtexs molt junts entre sí.

Aquí no té gaire sentit analitzar que els atributs es preservin bé, ja que encara que té una textura aquesta és molt simple, sent completament de color verd. Les normals sembla que estan correctes.

Pel que fa a la forma i els detalls, ara sí que veiem pèrdues importants. En els dos primers LODs hi ha moltes fulles que **es tornen petites o desapareixen completament**, tot i que a simple vista no es nota perquè segueix havent molts elements junts. El tercer LOD és crític, perquè la majoria de les fulles que sobreviuen **acaben ajuntant-se**, fent que es vegin amb un grossor elevat. Tot i així és un resultat molt acceptable ja que a l'últim LOD s'ha eliminat el 97 % dels vèrtexs. He fet un simplificació extrema justament per forçar i veure com es perden completament les formes.

Malgrat això, i tenint en compte que estem parlant d'un tros d'herba, l'últim LOD es podria utilitzar perfectament en un videojoc. L'herba sol ser un element molt petit, que està als peus del jugador i casi ningú si fixa, cosa que el fa raonable de simplificar en lo màxim possible.

## 5.6 Altres models

A continuació mostro les simplificacions de la resta dels models que tinc, amb les millors configuracions trobades. No necessiten gaires comentaris perquè tenen característiques similars als altres models que ja hem vist:

**Cat. (VC + QE)** Nota: els bigotis desapareixen o s'ajunten.



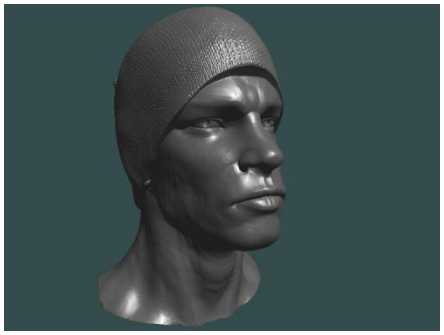
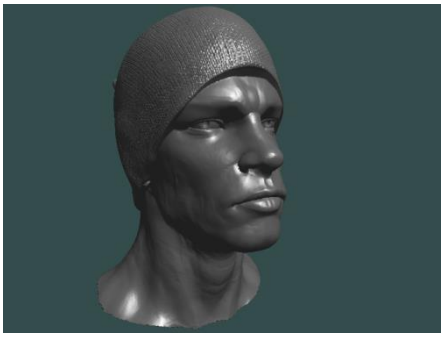
*LOD 1* *LOD 2* *LOD 3*

**Colonial. (VC)** Nota: molts meshes. Si t'apropes, es veu com poc a poc perden la connexió.



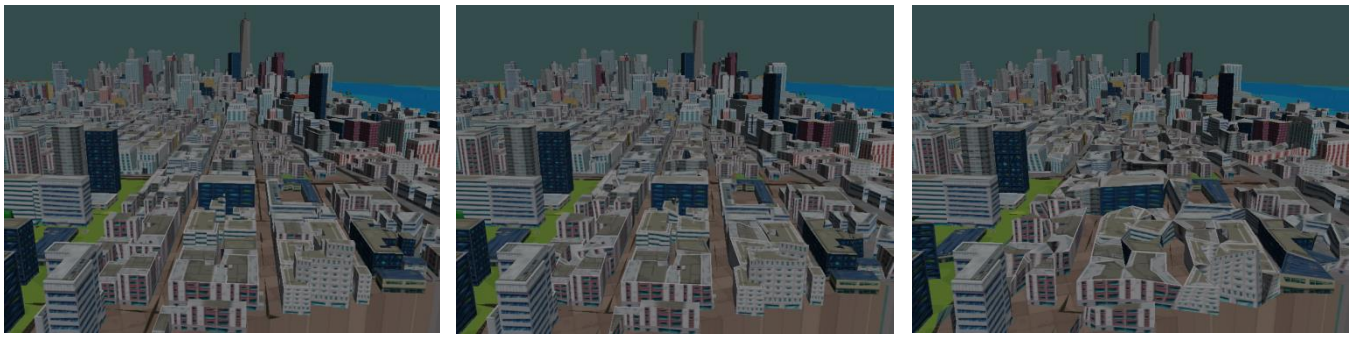
*LOD 1* *LOD 2* *LOD 3*

**Head. (VC + QE).** Nota: molt detall a la gorra. Taxa de reducció de dades enorme.


*LOD 1* *LOD 2* *LOD 3*

**NewYorkCity. (VC).** Nota: funciona fatal per QE. Edificis poligonals implica que es deformin al final.

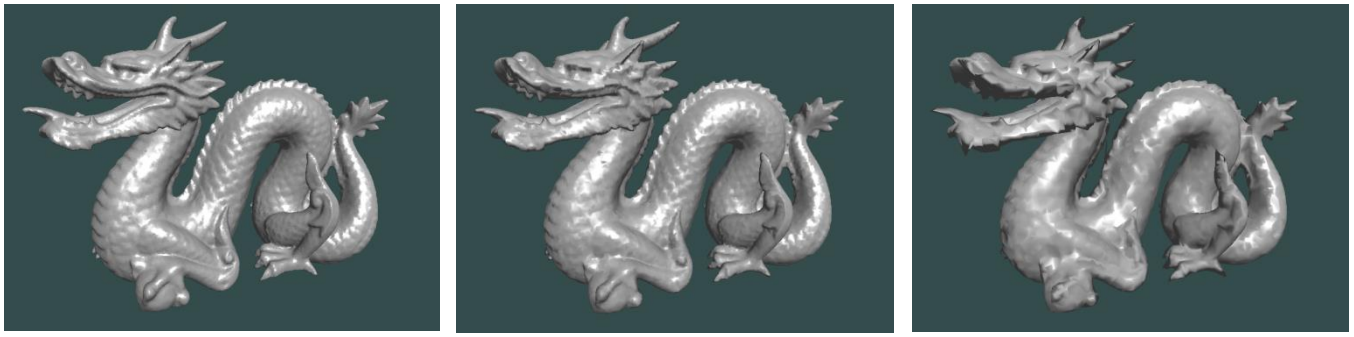


*LOD 1* *LOD 2* *LOD 3*

**Predator. (VC + QE).** Nota: Funciona molt bé amb Quadric Errors.

*LOD 1* *LOD 2* *LOD 3*

**Dragon. (VC + QE).** Nota: L'últim LOD té només el 2% de la geometria original, tot i així segueix mantenint la forma increïblement bé.



*LOD 1* *LOD 2* *LOD 3*

## 5.7 Unity

Per acabar amb les proves realitzaré un experiment pràctic amb els LODs obtinguts just abans. Carregaré aquests LODs en Unity per comparar rendiments, i a més implementaré un procés senzill per renderitzar el LOD que toqui en funció de la distància, mentre el joc s'està executant.

Unity és un motor de videojocs molt famós, completament gratuït i ràpid d'aprendre a utilitzar, perfecte per realitzar aquestes proves. No cal allargar més aquesta explicació amb les opcions que ofereix Unity, senzillament he creat un escena completament buida i he anat afegint models. Aquest són els experiments:

**Experiment 1**

El primer objectiu és veure quants models complexos pot aguantar Unity abans de començar a baixar el seu rendiment. Per mesurar el rendiment em fixo en els FPS de l'escena, és a dir, els fotogrames per segon. Normalment el joc s'executa ha una certa quantitat fixa de FPS, en aquest cas **90 FPS**, però a mesura que vaig afegint complexitat geomètrica a l'escena té més treball per renderitzar i arribarà un moment en el que no pugui calcular tot a la mateixa velocitat i per tant els FPS es redueixin.

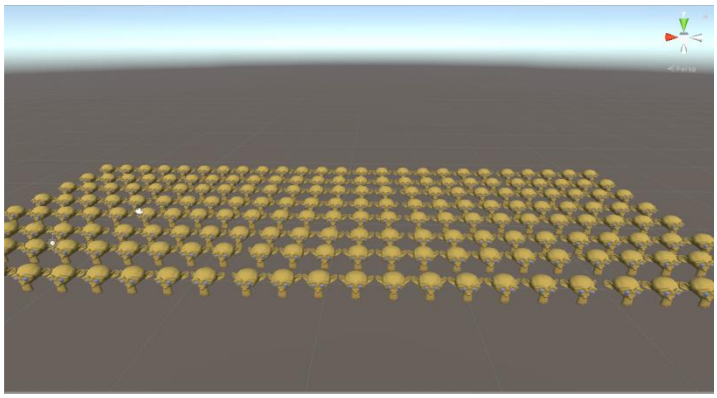
Veurem com evolucionen els FPS a mesura que afegim més models, i provant amb tots els LODs. Mentre els FPS es mantinguin **per sobre de 60**, es considera que el joc es veu ben fluid. Per **sobre de 30** és lo mínim acceptable que tenen els jocs avui en día.

Per al model del **Monkey** he obtingut els següents resultats. El LOD 0 és el model original, la resta de LODs són els obtinguts en la secció anterior. Els FPS són mitjanes:

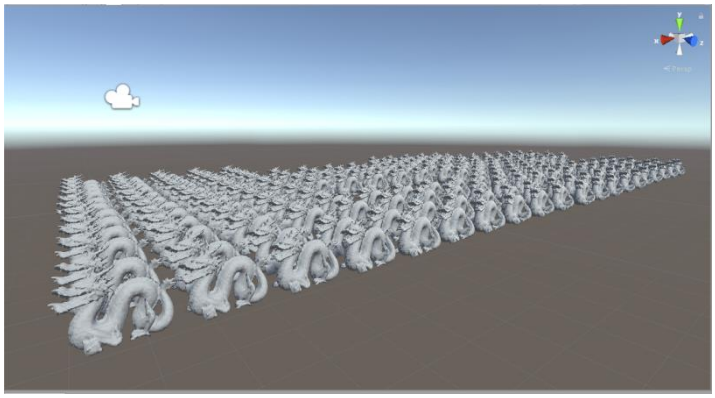
| | Quantitat de Models | | | | |
|---|---|---|---|---|---|
| **Monkey - FPS** | **10** | **50** | **100** | **150** | **200** |
| **LOD 0** | 90 | 40 | 20 | 15 | 10 |
| **LOD 1** | 90 | 90 | 55 | 32 | 20 |
| **LOD 2** | 90 | 90 | 90 | 75 | 55 |
| **LOD 3** | 90 | 90 | 90 | 90 | 90 |

Per al model del **Dragon** (més complex) he obtingut els següents resultats:

| | Quantitat de Models | | | | |
|---|---|---|---|---|---|
| **Dragon - FPS** | **10** | **50** | **100** | **150** | **200** |
| **LOD 0** | 60 | 19 | 5 | 3 | - |
| **LOD 1** | 90 | 43 | 20 | 14 | 9 |
| **LOD 2** | 90 | 90 | 68 | 40 | 33 |
| **LOD 3** | 90 | 90 | 90 | 90 | 90 |



*Escena amb 200 Monkeys*



*Escena amb 200 Dragons*

La millora és molt notòria. Clarament els models originals contenen massa geometria per ser tractada a partir d'un cert nombre de repeticions. A les imatges d'amunt estan renderitzades les versions més simplificades dels dos models, i **casi ni es nota la pèrdua del detall, però la millora del rendiment és espectacular**.

El cas més extrem és amb el model del drac. Renderitzant el LOD 0 per 200 vegades anava tant malament que ni tan sols em deixava executar el joc. En canvi amb el LOD 3 (que recordem, conté només el 2% de la geometria original) puc executar amb el màxim rendiment possible sense problemes, amb 90 FPS fixats.

Per aquest cas he seguit afegint més còpies del LOD 3 del drac per veure fins a on podia arribar. A partir de les 500 còpies més o menys ja començava a baixar dels 90 FPS i a amb 800 còpies baixava dels 60 FPS òptims. **Per baixar dels 30 FPS han calgut ni més ni menys que 1600 còpies, mentre que per baixar dels 30 FPS amb el LOD 0 es necessitaven unes 30 còpies**.

La conclusió d'aquest experiment és que efectivament la simplificació dels models aporta una gran millora en el rendiment de les escenes. La generació de LODs hauria de ser un element indispensable per qualsevol videojocs o aplicació gràfica en general en els que s'hagi de tractar amb models complexos.

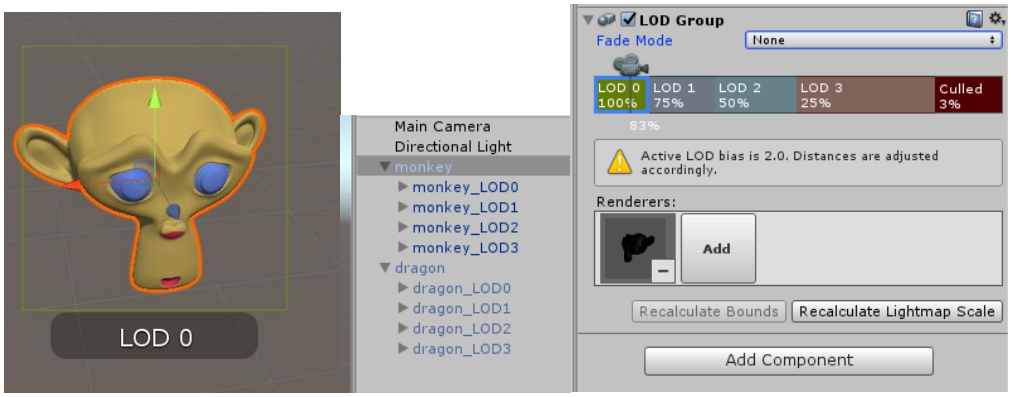
**Experiment 2**

Ara serà interessant centrar-se en el canvi visual que tenen els LODs quan substitueixen els models originals, en comptes del rendiment que es guanya.

Normalment quan estàs jugant un joc i t'apropes a un model complex, és correcte que aquest model es renderitzi amb el màxim detall possible. Ara bé, un cop t'allunyes no té cap sentit que tota aquesta geometria segueixi consumint recursos, per tant l'objectiu serà substituir els models complexos per les seves versions simplificades en funció de la distància de la càmera, i veure si aquest canvi visual es nota molt o passa desapercebut (que seria lo ideal).

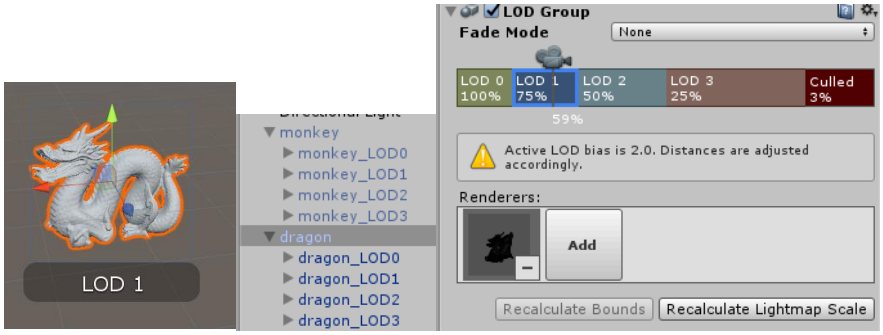
Per a fer això aprofito una opció que ofereix Unity per renderitzar LODs: **LOD Group**. Aquesta component l'afegeixes a un **gameObject** que conté tots els LODs com a fills, i et deixa assignar un mesh per cada LOD.

Seguidament pots determinar una **distància llindar** on cada LOD serà activat: en aquest exemple el LOD 0 és renderitzat des de el 100% fins el 75%, on es desactivarà i passarà a renderitzar-se el LOD 1. Aquests percentatges són mesurats a partir de la proporció entre l'altura de l'objecte i l'altura de la càmera. L'últim nivell, anomenat **Culled**, representa una distància tan allunyada on pràcticament l'objecte es veu com un píxel, i per tant ni cal renderitzar-lo (desapareix).

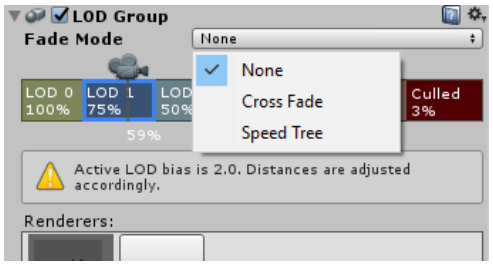



Un cop tens aquestes distàncies determinades ja està tot configurat i llest, així de ràpid i senzill. Des de dins del joc o simplement des del propi editor, a mesura que allunyes la càmera el model va canviat automàticament de LOD.

He provat el LOD Group amb el Monkey i el Dragon, i els resultats són molt satisfactoris tal i com esperava. La transició entre cada LOD és totalment imperceptible, inclús posant els llindars ben alts.




En el cas de que es veies el canvi visual, el LOD Group també ofereix una opció per realitzar les transicions més suaument: **Fade Mode**. Aquí pots escollir un dels dos modes, **Cross Fade o Speed Tree**, que serveixen per calcular el **factor de barreja**. Aquest factor s'utilitza per barrejar dos LODs consecutius, per tal de donar un efecte més suau de canvi. Per al Cross Fade senzillament et demana una amplada de transició, i el Speed Tree utilitza un shaders intern amb la variable "unity_LODFade". Però amb els meus exemples realment ni ha calgut utilitzar aquestes opcions.



## 5.8 Treball futur

El treball final de grau té un temps limitat, per aquesta raó he hagut de retallar algunes coses pensades inicialment.

He assolit tot el que m'havia proposat com a objectius: he creat una aplicació que és capaç d'importar qualsevol model, simplificar-lo de manera personalitzada, i tornar a exportar-lo per a la posterior utilització en videojocs o altres programes. També he investigat diferents tècniques de simplificació, havent implementat dues que han produït resultats molt satisfactoris (Vertex Clustering i Quadric Errors).

Tot i així, hi ha alguns aspectes que han quedat com a possible treball futur, que no són completament necessaris per aquest projecte però complementen i milloren el resultat obtingut:

- **Implementació de Edge Collapse**. Amb Vertex Clustering ja puc simplificar qualsevol model amb resultats altament satisfactoris, però per afegir més personalització Edge Collapse seria una molt bona opció. És complex i no accepta tots els models, però per aquells que funciona preserva millor els detalls.
- **Implementació de Normal Mapping**. Gran millora per aplicar a les simplificacions ja calculades, afegint detall a les textures sense incrementar la geometria. Aquesta tècnica completa el procés ideal d'optimització de models.
- **Creació d'una web per la meva aplicació**. En aquest document he proporcionat els links per descarregar l'aplicació, però seria més elegant i professional penjar SceneOptimizer de manera pública en alguna web. Aquesta web podria contenir informació / fòrums, a més d'alguna opció per acceptar donacions (mencionat més endavant per el pressupost del projecte).
- **Implementació d'un importador propi**. Ara mateix depenc completament d'ASSIMP per importar i exportar models, però resulta que va força lent i no acaba d'estar del tot optimitzat. De fet el coll d'ampolla de l'aplicació és importar un model: la resta d'algoritmes i simplificacions van molt més ràpid. Per aquest raó una bona opció seria implementar importadors propis, encara que impliqués molta feina per tenir en compte diferents formats.
- **Wireframe color.** Quan es renderitza el Wireframe és de color gris, i ja està bé per la majoria de models amb textura. El problema és quan importem un model sense textura, per tant gris, i provoca que casi no es vegi el Wireframe.
- **Tractament amb models extremadament complexos.** Amb la aplicació pots carregar tot tipus de models, amb complexitats molt diverses. Però té un límit de capacitat, i arriba un punt que deixa de funcionar si carregues models amb molta geometria. És l'exemple del model **Lucy** de la web de Stanford, que té 14 milions de vèrtexs i 28 milions de índexs, i ASSIMP es queda completament

sense memòria a l'intentar importar-lo. Normalment aquests models són casos extrems, i casi mai caldrà tractar amb ells, però és una llàstima que l'objectiu d'aquest projecte sigui simplificar models complexos i justament els més complexos no els puguem ni importar (la principal limitació és d'ASSIMP). Per a models amb 4 o 5 milions de vèrtexs i triangles encara funciona.

- **Petites millores de simplificació.** Els resultats amb Vertex Clustering i Quadric Errors són molt satisfactoris, però hi ha petites millores que encara es podrien aplicar, com evitar que els bigots del gat s'ajuntin o que diversos meshes perdin la connexió entre sí. Tot i així aquests esdeveniments casi ni es noten, només quan t'apropes molt, però en aquests casos interessa renderitza el model complex com hem vist en les proves de Unity.

Amb això acabo la redacció del contingut tècnic del meu projecte.

Com a petita conclusió final, recalcar que estic molt satisfet amb el resultat, sobretot tenint en compte que treballar amb models en general és arriscat. Cada model és un món, i que un algoritme funcioni bé per un cas pot ser un desastre per altres casos. De fet al principi vaig tenir bastants problemes per posar tot en marxa, per configurar correctament QT, ASSIMP i totes les eines que utilitzo. Tenint en compte això els resultats són molt motivadors. També he de donar les gràcies al meu director, que m'ha ajudat molt sobretot en aquells moments on estava encallat.

# Annexos

A continuació detallaré tots els recursos utilitzats durant el desenvolupament d'aquest projecte. Explicaré com he aconseguit tots els models de prova i les eines i llibreries necessitades. Tots els models estan disponibles a la carpeta Models.

## 6.1 Models utilitzats

**Monkey**

**Font:** L'he creat jo mateix amb Blender. Texturitzat de manera senzilla.

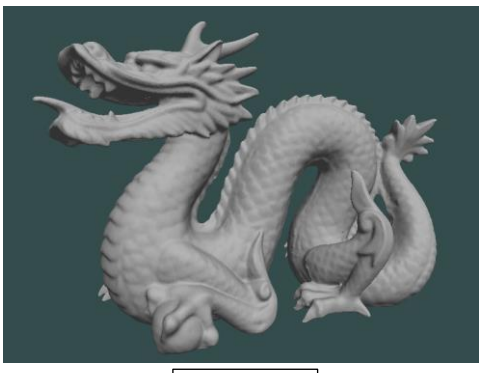
**Nanosuit**

**Font:** Model de test en LearnOpenGL. Link: https://sketchfab.com/models/ca311b94a0c249a1abc6697d105253e5

**Armadillo**

**Bunny**

**Models de Stanford.** Tots agafats del repositori de Stanford. Link: http://graphics.stanford.edu/data/3Dscanrep/



**Dragon**

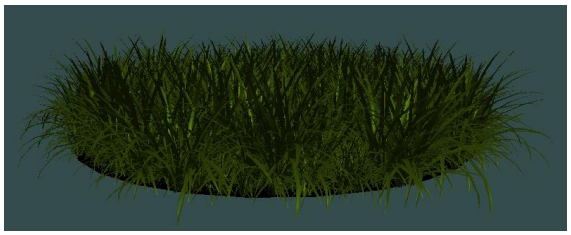
**xyzrgb_Dragon**

**Happy_Buda**

**Models de Turbosquid.** Models gratuïts d'aquesta web. Link: https://www.turbosquid.com/

**Colonial**

**Grass**



**Head**

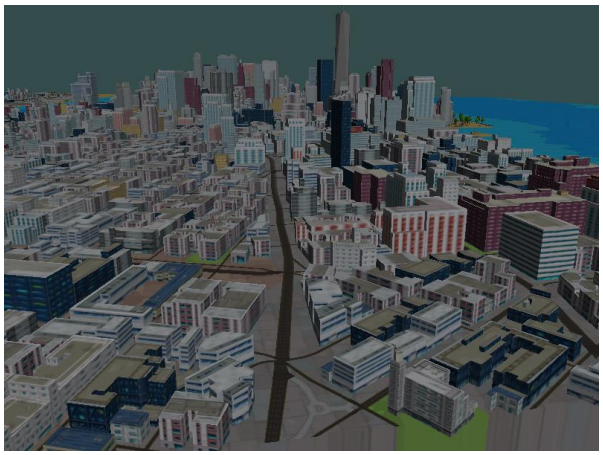
**Predator**

**Trees**

**Models de Free3D.** Models gratuïts d'aquesta web. Link: https://free3d.com

**Cat**

**New_york_city**



## 6.2 Eines i llibreries

Conjunt de tot el software utilitzat per el desenvolupament d'aquest projecte:

- **QT Creator 4.6.0, amb QT 5.10.1**. Entorn per desenvolupar el codi.
- **QT Designer 4.6.0**. Entorn per desenvolupar la interfície gràfica.
- **Core-Profile OpenGL 3.3**. Llibreria gràfica per la renderització de models 3D.
- **Compilador MinGW 32-bit**. Compilador utilitzat per QT.
- **Microsoft Visual Studio 2017 15.5.6**. Entorn per desenvolupar el codi.
- **CMake 3.10.2**. Programa per compilar llibreries estàtiques i dinàmiques.
- **ASSIMP 4.1.0**. Llibreria per importar i exportar els models.
- **Eigen 3.3.4**. Llibreria matemàtica de gran precisió per àlgebra lineal.
- **GLM 0.9.9**. Llibreria matemàtica per treballar amb vectors i matrius.
- **Blender 2.77**. Programa de modelatge 3D.
- **Unity 2017.3**. Motor de videojocs per fer proves.
- **XNormal 3.19.3**. Programa per generar automàticament Normal Maps.
- **Github**. Per guardar el codi.
- **Mega**. Per poder descarregar el meu programa.
- **Trello**. Per el correcte seguiment de les tasques.

El desenvolupament l'he realitzat en aquests dos ordinadors:

| | **Acer Predator G3620 (PC)** | **MSI GL62M (Portàtil)** |
|---|---|---|
| **OS** | Windows 10 (64 bits) | Windows 10 (64 bits) |
| **CPU** | Intel Core i7-3770 (3.40 GHz) | Intel Core i7-7700 (2.8 GHz) |
| **GPU** | GeForce GTX 970, 4GB GDDR5 | GeForce GTX 1050, 4GB GDDR5 |
| **RAM** | 16GB DDR3 | 8GB DDR4 |

# GEP

A continuació es mostren els continguts treballats durant el curs de Gestió de Projectes. Algunes parts ja les he explicat en el transcurs del document, com els objectius i els recursos utilitzats, així que en aquesta secció em centraré principalment en la gestió temporal, la gestió econòmica i l'anàlisi de sostenibilitat.

## 7.1 Gestió Temporal

### 7.1.1 Actors implicats en aquest projecte

- **Desenvolupador**. Jo, autor del projecte, sóc l'únic desenvolupador. A més de servir-me per acabar el grau, aquest projecte també em beneficiarà per aprendre OpenGL i aprofundir en el món del gràfics per computador, a més de que la aplicació final em serà útil si algun cop em trobo que necessito tractar amb models complexos.
- **Director**. El director del projecte s'encarrega de guiar-me, supervisant les entregues i assegurant que compleixo amb els objectius, podent també guiar-me en del desenvolupament si em quedo encallat. També em podrà proporcionar models interessants per fer proves.
- **Usuaris**. Qualsevol persona que utilitzi SceneOptimizer per optimitzar els seus models. És una aplicació gràfica gratuïta (per PC / MAC) i no hi ha cap requisit per ser utilitzada.
- **Modeladors / Artistes**. També és important tenir en compte totes aquelles persones que modelen els models. Evidentment jo no sóc artista, per tant tots els models que utilitzi per fer les proves són descarregats o obtinguts d'aquestes persones que en saben més.

### 7.1.2 Definició de l'Abast

Aquest és un projecte complex que s'ha de realitzar en un temps relativament curt, per tant cal tenir clar el camí que se seguirà, fins a on es podrà arribar i com es planificarà el procés de desenvolupament. A partir dels objectius defineixo les següents fites:

1. El primer és tenir un sistema per tractar de manera externa amb els models. Un mètode per poder importar i exportar cap a la meva aplicació, amb l'ajuda **d'ASSIMP**. Sense això no podré avançar ni implementar res més, per tant és un pas molt important que funcioni bé.
2. El següent és investigar possibles **tècniques de simplificació** de models i triar la millora per implementar-la.
3. Seguidament arriba la secció de **desenvolupament**, per implementar el que he investigat i seleccionat. Per fer proves senzilles no calen models complexos.
4. De manera paral·lela també caldrà anar dissenyant la **interfície gràfica** de l'aplicació, amb les funcionalitats que vagi afegint poc a poc.
5. Un cop acabat caldrà fer moltes **proves** amb tot tipus de models.
6. Finalment quedarà la part per redactar la **memòria** explicant tot el que he fet, ensenyant els resultats de totes les proves realitzades.

Un cop completades aquestes fites el resultat ja és adequat per a aquest projecte. No cal que l'aplicació final funcioni perfecte, ni que sigui capaç de simplificar perfectament tots els models existents, ja que és un tema complicat i crític. Lo important és que sigui lo més robust possible, eficient i fàcil d'utilitzar per qualsevol usuari.

Alguns dels possibles obstacles que hem puc trobar durant tot el procés:

- **No ser capaç d'importar amb ASSIMP**. De ser així, tocaria implementar un importador propi per tractar com a mínim amb el format .obj (Wavefront Object), que és el més comú. Això em suposaria temps addicional en la planificació inicialment pensada.
- **Que apareguin errors en el codi**. Serà important tenir la aplicació gràfica implementada lo abans possible, per poder solucionar els errors de manera més fàcil i còmode.
- **Models incorrectes**. Si els models estan mal modelats no és culpa meva, per tant no puc fer res per solucionar-ho. Senzillament caldrà buscar altres models.

Durant el desenvolupament seguiré una metodologia àgil. El propòsit és tenir sempre un programa funcional sobre el que vaig afegint més funcionalitats però que per si sol sigui correcte. Així es pot ensenyar en les reunions setmanals amb el director. La planificació serà estricte i les entregues concretes.

### 7.1.3 Definició de les tasques

El projecte dura aproximadament 4 mesos, del 19 de freber que comença el quadrimestre, fins al 25 de juny que és la setmana de les presentacions del TFG.

A partir dels objectius i l'abast he definit les següents tasques per realitzar durant aquest temps:

1. Fita Inicial
    a. Abast i Contextualització
    b. Planificació Temporal
    c. Gestió Econòmica i Sostenibilitat
    d. Presentació Preliminar
    e. Revisió de les Competències
    f. Presentació de la Fita Inicial
2. Desenvolupament
    a. Prepara entorn i llibreries
    b. Configurar ASSIMP
    c. Investigar tècniques de simplificació
    d. Implementar Vertex Clustering
    e. Implementar Quadric Errors
    f. Construir aplicació gràfica
3. Proves amb tots tipus de models
4. Redacció de la memòria
5. Presentació Final

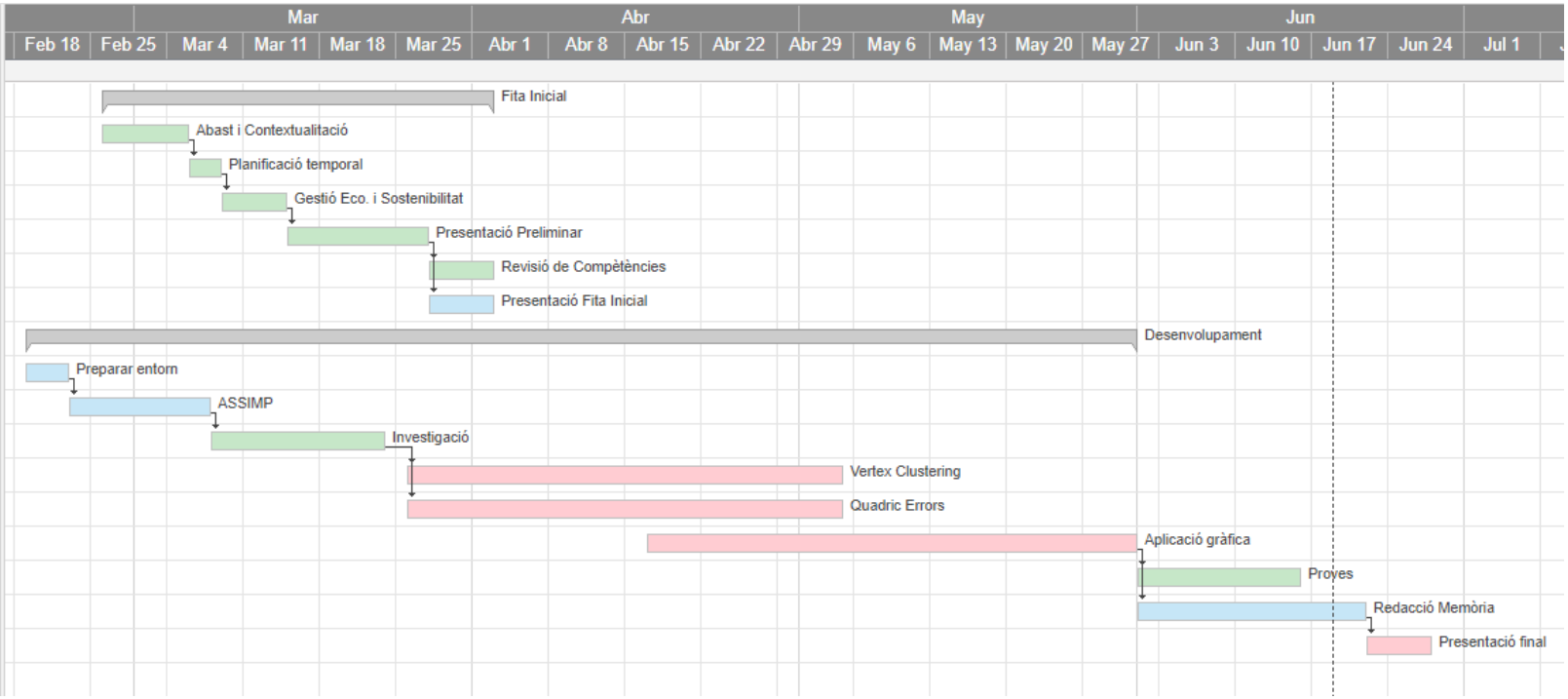
El significat de cada tasca ja ha estat explicat amb anterioritat. Les dependències estan ben explicades en el diagrama de Gantt. En aquesta llista no he inclòs les reunions amb el director, però seran més o menys cada setmana i mitja o cada dues setmanes.

### 7.1.4 Duración de les tasques

| Tasca | Duració | Risc |
|---|---|---|
| **Fita Inicial** | **75** | **Baix** |
| **Desenvolupament** | **310** | |
| **Preparar entorn i llibreries** | 30 | Normal |
| **ASSIMP** | 30 | Normal |
| **Investigació de tècniques** | 50 | Baix |
| **Vertex Clustering** | 60 | Alt |
| **Quadric Errors** | 60 | Alt |
| **Aplicació gràfica** | 80 | Alt |
| **Provar i resultats** | **40** | **Baix** |
| **Redacció Memòria** | **40** | **Normal** |
| **Presentació final** | **10** | **Alt** |
| **Total** | **475** | |

### 7.1.5 Diagrama de Gantt

| Tasca | Inici | Final | Rol | Risc |
|---|---|---|---|---|
| Fita Inicial | 26/02/18 | 02/04/18 | | |
| Abast i Contextua | 26/02/18 | 05/03/18 | Director | Baix |
| Planificació temp | 06/03/18 | 08/03/18 | Director | Baix |
| Gestió Eco. i Sos | 09/03/18 | 14/03/18 | Director | Baix |
| Presentació Preli | 15/03/18 | 27/03/18 | Director | Baix |
| Revisió de Comp | 28/03/18 | 02/04/18 | Director | Baix |
| Presentació Fita I | 28/03/18 | 02/04/18 | Director | Normal |
| Desenvolupament | 19/02/18 | 31/05/18 | | |
| Preparar entorn | 19/02/18 | 22/02/18 | Desenvolu | Normal |
| ASSIMP | 23/02/18 | 07/03/18 | Desenvolu | Normal |
| Investigació | 08/03/18 | 23/03/18 | Desenvolu | Baix |
| Vertex Clustering | 26/03/18 | 04/05/18 | Desenvolu | Alt |
| Quadric Errors | 26/03/18 | 04/05/18 | Desenvolu | Alt |
| Aplicació gràfica | 17/04/18 | 31/05/18 | Desenvolu | Alt |
| Proves | 01/06/18 | 15/06/18 | Beta Teste | Baix |
| Redacció Memòria | 01/06/18 | 21/06/18 | Director | Normal |
| Presentació final | 22/06/18 | 27/06/18 | Director | Alt |




Verd → Risc Baix
Blau → Risc Normal
Vermell → Risc Alt

Rols: Director, Desenvolupador i Beta Tester. Es tindran en compte per el pressupost.

## 7.2 Gestió Econòmica

En aquesta secció es farà una estimació dels costos d'aquest projecte, tenint en compte els costos dels recursos humans, despeses indirectes, hardware i software.

L'objectiu d'aquest apartat és construir un pressupost per al projecte, seguint la planificació (basat en el Gantt) i els recursos a utilitzar.

### 7.2.1 Recursos Humans

Aquest projecte té un únic autor, jo mateix, que m'encarregaré de tots els rols. Seguidament especifico breument quins són aquests rols, i el seu valor econòmic que representen (tenint en compte les hores invertides).

| Rol | Hores | Preu | Cost |
|---|---|---|---|
| Director | 75 | 40 €/h | 3.000 € |
| Desenvolupador | 310 | 20 €/h | 6.200 € |
| Beta Tester | 90 | 10 €/h | 900 € |
| **Total** | **475** | | **10.100 €** |

Els preus s'han determinat tenint en compte els sous reals actuals a Espanya i el temps estimat per cada rol.

### 7.2.2 Despeses Indirectes

Aquestes són les despeses derivades principalment de l'electricitat consumida. El projecte té una duració de 475 hores i els 2 ordinadors tenen una potència mitja de 86W (no els usaré a la vegada).

| Producte | Preu | Unitats | Cost aproximat |
|---|---|---|---|
| Electricitat | 0,14 €/kWh | 41 kWh | 5,74 € |
| **Total** | | | **5,74 €** |

### 7.2.3 Hardware

Totes les tasques del projecte les realitzaré en els dos ordinadors que he comentat en la secció de recursos de la planificació. Cal tenir en compte la seva amortització (suposant que el projecte dura 4 mesos).

| Producte | Preu | Vida útil | Amortització |
|---|---|---|---|
| Acer Predator (PC) | 1.053 € | 6 | 58,5 € |
| MSI (Portatil) | 999 € | 5 | 66,6 € |
| **Total** | **2.052 €** | | **125,1 €** |

### 7.2.4 Software

També cal mencionar totes les eines software que utilitzaré (i les seves llicències). És notable que totes són Open Source o gratuïtes, per tant no hi haurà cap cost per part del software.

| Software | Preu | Amortització |
|---|---|---|
| OpenGL | 0 € | 0 € |
| ASSIMP | 0 € | 0 € |
| QT Creator | 0 € | 0 € |
| CMake | 0 € | 0 € |
| GLM, EIGEN | 0 € | 0 € |
| Blender | 0 € | 0 € |
| Unity | 0 € | 0 € |
| Github, Trello, SmartSheet, RefWorks | 0 € | 0 € |
| **Total** | **0 €** | **0 €** |

### 7.2.5 Pressupost total

| Concepte | Cost | Amortització |
|---|---|---|
| Recursos Humans | 10.100 € | 10.100 € |
| Despeses Indirectes | 5,74 € | 5,74 € |
| Hardware | 2.052 € | 125,1 € |
| Software | 0 € | 0 € |
| **Total** | **12.157,74 €** | **10.230,84 €** |

Aquest és el pressupost final. Sortiria per uns 2.558 € al més, durant els 4 mesos que dura el projecte. És força car, sobretot tenint en compte que l'aplicació final és Open Source, per tant gratuïta, i no s'obtindran beneficis. Una alternativa per fer-lo més viable seria afegir l'opció d'enviar donacions.

### 7.2.6 Control de Desviacions

Addicionalment sempre s'ha de considerar l'aparició d'imprevistos i desviacions, lo que pot comportar un augment dels costos. Alguns d'aquests imprevistos amb els seus plans d'acció són:

**Espatllament del hardware**. Com que utilitzo dos ordinadors, no seria massa crític, però en el cas que s'espatllessin els dos llavors hauria de comprar un PC nou (o si és possible, reparar-los, però moltes vegades surt encara més car).

**Tasca que s'allarga**. Serà imprescindible reorganitzar el diagrama de Gantt i redistribuir els recursos i temps que queda. Si inicialment he calculat malament el cost d'una tasca important i dura més del pensat, hauré de retallar d'altres tasques menys importants.

Tot el pressupost està calculat a partir de les tasques presentades al diagrama de Gantt, però sempre s'intentarà ajustar als recursos disponibles en cada moment.

Les desviacions en el cost final es podran calcular amb les fórmules de desviacions proporcionades per el curs de GEP.

## 7.3 Anàlisi de Sostenibilitat

Construcció de la matriu de sostenibilitat:

### 7.3.1 Projecte Posat en Producció

| Dimensió | Reflexió |
|---|---|
| **Econòmica** | El cost estimat per al projecte és una mica elevat tenint en compte que el producte final serà gratuït, i no s'obtindran beneficis. Una opció seria afegir la possibilitat pels usuaris de poder enviar donacions. |
| **Ambiental** | L'impacte ambiental d'aquest projecte és l'estimat per a qualsevol projecte similar: la petjada mediambiental d'obtenir l'electricitat i la petjada ecològica de la fabricació del hardware (explotació del medi per obtenir els materials, etc). No m'he plantejat minimitzar-ne l'impacte, ja que és força reduït. |
| **Social** | A nivell personal aquest projecte és una gran oportunitat per aprendre OpenGL a fons i incrementar els meus coneixements en el món dels gràfics per computador, el qual em sembla molt interessant. |

### 7.3.2 Vida Útil

| Dimensió | Reflexió |
|---|---|
| **Econòmica** | Actualment aquest problema està implementat en alguns motors gràfics o de modelatge 3D, que poden tenir una llicencia de pagament o ser gratuïts, acceptant també donacions. La meva llibreria serà Open Source, per tant qualsevol usuari la podrà usar sense preocupar-se pel preu. |
| **Ambiental** | El problema a abordar és purament de software, per tant no té cap implicació directa mediambiental. Com que no hi ha res a resoldre en aquest aspecte, la meva solució no millorarà res respecte les existents. |
| **Social** | Actualment hi ha motors gràfics que apliquen aquest problema, però no tenen flexibilitat per escollir els paràmetres ni per tractar tot tipus d'escenes i models variats. Això crea la necessitat real de disposar d'una llibreria completament independent de qualsevol altra programa, per facilitar l'optimització adient a qualsevol model, poden escollir els paràmetres personalitzats. Qualsevol usuari la podrà usar, sense la necessitat d'aprendre un motor gràfic nou ni res addicional. |

### 7.3.3 Riscos

| Dimensió | Reflexió |
|---|---|
| **Econòmica** | Econòmicament no té riscos reals, ja que en realitat no es pagarà a ningú. Si traiem els hipotètics costos de Recursos Humans, la resta de costos són negligibles (despeses indirectes i hardware). |
| **Ambiental** | El problema a abordar és purament de software, per tant no té cap implicació directa mediambiental. |
| **Social** | El producte final no hauria de suposar cap risc per als usuaris, en tot cas l'únic que pot fer és millorar el seu treball amb models 3D. |

### 7.3.4 Puntuació final Sostenibilitat

| | PPP | Vida Útil | Riscos |
|---|---|---|---|
| **Econòmic** | 5 | 8 | -2 |
| **Ambiental** | 4 | 10 | -2 |
| **Social** | 10 | 15 | -2 |
| **Rang Sostenibilitat** | 19 | 33 | -6 |
| **Rang Total** | | **46** | |

# Bibliografia

## Papers


[1] Kok-Lim, L. ; Tiow-Seng, T. (2001). *Model simplification using vertex-clustering*. [en línia]. National University of Singapore. [Consultat: 2018]. Disponible a internet: < https://www.comp.nus.edu.sg/~tants/Paper/simplify.pdf >

[2] Hoppe, H. (1996). *Progressive meshes*. [en línia]. Microsoft Research. [Consultat: 2018]. Disponible a internet: < http://hhoppe.com/pm.pdf >

[3] Willmott, A. (2011). *Rapid simplification of multi-attribute meshes*. [en línia]. Maxis/Electronic Arts. [Consultat: 2018]. Disponible a internet: < http://www.andrewwillmott.com/papers/rsmam >

[4] Tarini, M. ; Cignoni, P. ; Scopigno, R. (2003). *Visibility based methods and assessment for detail-recovery*. [en línia]. Istituto Di Scienza e Tecnologie dell'Informazione. [Consultat: 2018]. Disponible a internet: < http://vcg.isti.cnr.it/~tarini/pap0A/vis03.pdf >

[5] Garland, M. ; Heckbert, P. (1997). *Surface Simplification Using Quadric Error Metrics*. [en línia]. Carnegie Mellon University. [Consultat: 2018]. Disponible a internet: <https://www.ri.cmu.edu/pub_files/pub2/garland_michael_1997_1/garland_michael_1997_1.pdf >


## Llibres


[6] SELLERS, G. ; S. Wright, R. ; HAEMEL, N. (2015). *OpenGL Superbible: Comprehensive Tutorial and Reference*. 7ª ed. Addison-Wesley Professional.

[7] KESSENICH, J. ; SELLERS, G. ; SHREINER, D. (2016). *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. 9ª ed. Addison-Wesley Professional.

## Webs


[8] de Vries, J. (2017). *Learn OpenGL*. [en línia]. [Consultat: 2018]. Disponible a internet: < https://learnopengl.com/ >

[9] Kulling, K. (2018). *Open asset import library*. [en línia]. [Consultat: 2018]. Disponible a internet: < http://assimp.sourceforge.net/ >

[10] Segal, M. ; Akeley, K. (2010). *The OpenGL R graphics system: A specification*. [en línia]. [Consultat: 2018]. Disponible a internet: < https://www.khronos.org/registry/OpenGL/specs/gl/glspec33.core.pdf >

[11] Stanford (2018). *The Stanford 3D Scanning Repository*. [en línia]. Stanford Computer Graphics Laboratory. [Consultat: 2018]. Disponible a internet: < http://graphics.stanford.edu/data/3Dscanrep/ >

[12] CMSC (2013). *Quadric Error Metrics*. [en línia]. [Consultat: 2018]. Disponible a internet: <https://www.csee.umbc.edu/courses/graduate/635/spring13/lectures/05_QuadricErrorMetrics.pdf >

[13] Sau Yiu (1997). Mesh Simplification Using *Quadric Error Metrics*. [en línia]. [Consultat: 2018]. Disponible a internet: < https://classes.soe.ucsc.edu/cmps160/Spring05/finalpages/scyiu/ >

[14] OpenGL (2018). *Tutorial 13 Normal Mapping*. [en línia]. [Consultat: 2018]. Disponible a internet: < http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/ >

[15] Unity (2018). *Unity User Manual*. [en línia]. [Consultat: 2018]. Disponible a internet: < https://docs.unity3d.com/Manual/index.html >

[16] Blender (2018). *Blender Reference Manual*. [en línia]. [Consultat: 2018]. Disponible a internet: < https://docs.blender.org/manual/en/dev/ >

[17] xNormal (2018). *xNormal*. [en línia]. [Consultat: 2018]. Disponible a internet: < http://www.xnormal.net/ >

[18] QT (2018). *QT Documentation*. [en línia]. [Consultat: 2018]. Disponible a internet: < http://doc.qt.io/qt-5/modules.html?hsCtaTracking=d23dd544-ef00-4b60-a9b0-13e61b61b5b0%7C610cf161-29a9-4d45-b0d1-18ba07031258 >

[19] Github (2018). [en línia]. [Consultat: 2018]. Disponible a internet:
< https://github.com/ >

[20] Trello (2018). [en línia]. [Consultat: 2018]. Disponible a internet:
< https://trello.com/ >